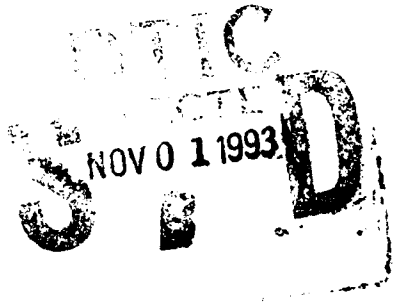
1993-06

Simulation of tripod gaits for a hexapod underwater walking machine

Schue, Charles Andrew, III

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

SIMULATION OF TRIPOD GAITS FOR A HEXAPOD UNDERWATER WALKING MACHINE

by

Charles Andrew Schue, III

June 1993

Thesis Advisor: Yutaka Kanayama

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approval for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) ECE	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION National Science Foundation 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

SIMULATION OF TRIPOD GAITS FOR A HEXAPOD UNDERWATER WALKING MACHINE

12. PERSONAL AUTHOR(S)

Schue, Charles Andrew III

13a. TYPE OF REPORT	13b. TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM TO	1993 June	270

16. SUPPLEMENTARY NOTATION

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Robotics, Walking Machines, Kinematics, AquaRobot

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis develops the mathematical relationships necessary to implement alternating tripod gaits on the hexapod underwater walking machine, *AquaRobot*. Analysis of documentation and application of Denavit-Hartenberg kinematic modeling techniques determine the fundamental vehicle parameters. Smooth leg motion models following elliptical and cycloidal trajectories are devised. Gait planning algorithms, using the elliptical smooth leg motion model, are developed for both discrete and continuous body motion. Statically stable, alternating tripod gait simulations are implemented in the C++ programming language. A stick figure graphics display allows examination and testing of the gait algorithms prior to incorporation in follow-on 3D graphics simulations or in real-time operation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Yutaka Kanayama	22b. TELEPHONE (Include Area Code) (408)656-2095 22c. OFFICE SYMBOL CS/Ka

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603 Unclassified

Approved for public release; distribution is unlimited.

SIMULATION OF TRIPOD GAITS FOR A HEXAPOD UNDERWATER WALKING MACHINE

by

Charles Andrew Schue, III

Lieutenant, United States Coast Guard
B.S., Naval Postgraduate School, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

June 1993

Author: [redacted]
Charles Andrew Schue, III

Approved by: [redacted]
Yutaka Kanayama, Thesis Advisor

[redacted]
Roberto Cristi, Second Reader

[redacted]
Michael A. Morgan, Chairman,
Department of Electrical and Computer Engineering

ABSTRACT

This thesis develops the mathematical relationships necessary to implement alternating tripod gaits on the hexapod underwater walking machine, *AquaRobot*. Analysis of documentation and application of Denavit-Hartenberg kinematic modeling techniques determine the fundamental vehicle parameters. Smooth leg motion models following elliptical and cycloidal trajectories are devised. Gait planning algorithms, using the elliptical smooth leg motion model, are developed for both discrete and continuous body motion. Statically stable, alternating tripod gait simulations are implemented in the C++ programming language. A stick figure graphics display allows examination and testing of the gait algorithms prior to incorporation in follow-on 3D graphics simulations or in real-time operation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

No thesis is written without the help of others. I would like to acknowledge those that made this project possible. In particular, I would like to express my gratitude to:

- Professor Yutaka Kanayama, for his counsel, guidance, and unwavering support throughout the project;
- Professor Roberto Cristi, for agreeing to be my second reader;
- Professor Robert McGhee, for starting me on the robotics path;
- Dwight Alexander, Sandra Davidson, Chuck Lombardo, and Kenji Suzuki for their comments, suggestions, and just for being friends;
- Mr. Mineo Iwasaki, Mr. Hideotashi Takahashi, and Mr. Shigeki Shiraiwa, at the Japan Port and Harbor Institute, for allowing me to visit their fine facility and for providing their significant technical support;
- Drs. Shigeo Hirose and Kan Yoneda, at the Tokyo Institute of Technology, for allowing me to partake in the wonders of their robotics program; and
- to the National Science Foundation which, under Grant BCS-9109989, supported a portion of the work in this thesis.

To each and all of you, my sincere thanks.

As much as I have invested in this thesis, my wife, Lori, and my children, Ian and Tirena, are still my best investment. Without them, there would be no reason to achieve. To my family: a special, loving, thank-you.

I. INTRODUCTION

Most of us are familiar with the development and widespread use of wheeled locomotion. Prepared surfaces, in the form of roadways and railways, provide the means for wheeled vehicles to efficiently transport people and property over long distances. Wheeled vehicles, however, are significantly less efficient over irregular, unprepared terrain. Even tracked vehicles, whose tracks are simply elongated wheels, have difficulty traversing irregular terrain. Wheeled and tracked vehicles, then, require *special* terrain.

Legged locomotion, on the other hand, offers several potential advantages over wheels or tracks when traversing natural terrain. First, and foremost, legs allow the flexibility to select and place supporting feet. Flexible foot placement provides opportunities to test the terrain prior to attempting vehicle support and extends vehicle mobility beyond prepared surfaces. Legged vehicles can move on more *general* terrain.

Another advantage is enhanced traction in pliable soil. Wheeled and tracked vehicles, as an inherent part of their design, make a depression in any non-supporting surface and then constantly try to climb out of that depression. Legged vehicles typically generate distinct footprints and use any material pushed up behind the foot to improve traction. [MCG85]

Additional potential advantages of legged locomotion over wheeled or tracked locomotion include greater speed, less power consumption, and relatively small footprint. Finally, adoption of legged locomotion offers the unique advantage of integrating the locomotion function with a terrain measurement function. Specifically, legged locomotion allows simultaneous vehicle support, propulsion, and measurement of terrain height or depth.

Legged locomotion advantages are not lost in an underwater environment. Of the vehicles capable of submersible operation, 90 percent are of the tethered or untethered free swimming type. Although these vehicles effectively use their three-dimensional maneuverability, they have difficulty generating significant forces and have trouble maintaining stationary position and direction. The underwater wheeled or tracked (crawling) vehicles have poor maneuverability, but are capable of exerting very large working forces. As in the non-aquatic case, underwater wheeled or tracked locomotion is limited to relatively hard and flat sea bottom. Additionally, wheeled or crawling vehicles make the water so muddy that optical viewing devices or television cameras are rendered useless. Even in an underwater environment, legged locomotion overcomes many of the disadvantages of wheeled, tracked, or free swimming type locomotion. Legged vehicles can walk on uneven terrain with minimal disturbance of the surface, can provide a maneuverable, yet stable, platform for observation purposes, and can use appendages for accurate measurement. [ISH83]

A. *AQUAROBOT* UNDERWATER WALKING MACHINE

In the early 1980's, the Japanese Port and Harbor Research Institute (PHRI) was tasked with finding an alternative method of carrying out inspection of underwater construction sites. They determined a robot was needed because of increased risks to and lower working efficiency of port construction workers at deeper sea areas and also because of a shortage of divers. Additionally, the PHRI decided the ten centimeter horizontal and five centimeter vertical accuracy requirements and average working depth of 50 meters were best achieved by mechanical means.

The *AquaRobot* project started in 1984. *AquaRobot* was originally designed with two primary functions: 1) measurement of the flatness of rock foundation mounds for tsunami

breakwaters using the motion of its six walking legs, and 2) observation and supervision of underwater construction using its on-board television camera. An experimental model and a prototype model were constructed. The experimental model was developed for above water research and testing. Based on the results of testing the experimental model, a watertight prototype model was subsequently developed.

AquaRobot (Figure 1-1) is a six-legged, "insect type" robot. The body is hexagonal and the legs are installed on the sides of the hexagonal frame. Each of its six legs has three articulated joints driven by DC motors. Each leg includes a touch sensor on the foot. *AquaRobot* is currently controlled by a 16 bit 80286-based microcomputer. Additional geometrical detail of *AquaRobot* is provided in Chapter III. A more complete overview and history is found in Appendix A. [AKI89]

B. MOTIVATION FOR *AQUAROBOT* GAIT PLANNING

This thesis is a direct outgrowth of a cooperative research effort between the Naval Postgraduate School (NPS) and the Japanese PHRI to enhance the hardware and software capabilities of the *AquaRobot* underwater walking robot. The National Science Foundation (NSF) is providing support to the NPS, while the Japanese Science and Technology Agency (STA) is supporting the PHRI.

The PHRI originally developed *AquaRobot* to replace hard-hat divers constructing tsunami barriers in the hazardous underwater environment off Japan's coast. Although *AquaRobot* has demonstrated significant ability as the first hexapod underwater walking machine, it is still unable to perform its designed task more efficiently or less costly than human divers. *AquaRobot* speed and agility enhancements derived from gait planning algorithms in this thesis will result in a more efficient and less costly machine. This translates directly into reduced risk of human injury and decreased construction costs.

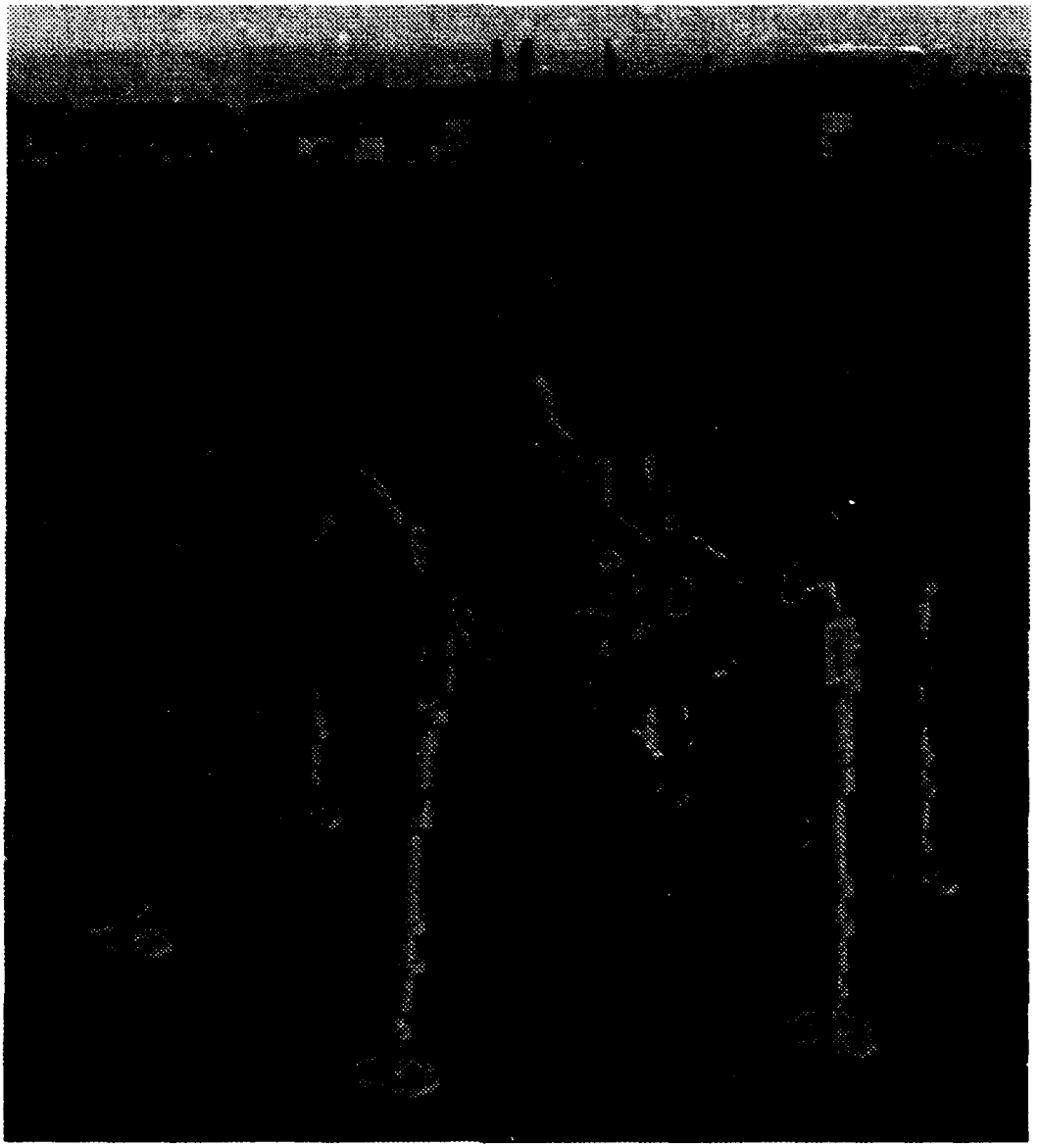


Figure 1-1. The Hexapod Underwater Walking Machine *AquaRobot*.

Although walking machine research and construction is extensive, *AquaRobot* is the first walking robot with a practical use. As such, improvements to *AquaRobot* directly result in concept fulfillment and favorably demonstrate the usefulness of robots, and walking robots in particular.

C. LITERATURE SURVEY

Obviously, the most reviewed literature was that generated at the PHRI by Akizono [AKI89] and Iwasaki [IWA87], [IWA88a], [IWA88b], and [IWA90]. These papers provided *AquaRobot*-specific development and implementation information. Craig's textbook [CRA86] yielded the fundamental robotics techniques for solutions of kinematics, inverse kinematics, coordinate transformations, and workspace problems.

Some information found in Hirose's seminal series of papers on gait algorithms for quadruped vehicles [HIR84], [HIR86a], [HIR86b], and [HIR88] proved transferable to hexapod vehicles. Two of McGee's papers [MCG85], [MCG79] were consulted for their insight into hexapod walking issues.

Although several walking robot textbooks were reviewed, specific walking theory was best presented in Song's textbook [SON89], while general walking theory was covered quite well in Todd's textbook [TOD85].

Lyman's thesis [LYM87], along with papers by Kwak [KWA90], Lee [LEE88a] and [LEE88b], and Waldron [WAL84] provided analysis of gait control for the Adaptive Suspension Vehicle, a hexapod walking machine with bilateral symmetry.

Davidson's [DAV93] and Grim's [GRI93] theses provided baseline graphics code development information. Kanayama's manuscript on spatial reasoning [KAN92] proved the best source for efficient motion planning algorithms.

D. THESIS ORGANIZATION

The final part of this chapter is devoted to outlining the thesis organization. Chapter II provides the setting for the remainder of the thesis. A brief history of walking machines is presented, along with a survey of previous work and an introduction to walking theory.

A physical description of the *AquaRobot* underwater walking robot is the emphasis of Chapter III. Geometrical features and system parameters are defined. Some aspects of *AquaRobot* are idealized for simplification. A detailed problem statement is included in this chapter.

Existing *AquaRobot* documentation is sparse and most detailed information is written in Japanese. Therefore, Chapter IV develops fundamental vehicle parameters through analysis of written reports, review of construction blueprints, and application of Denavit-Hartenberg kinematic modeling techniques. Inverse kinematic equations and relations are presented and necessary coordinate transformations are developed. Workspace volumes are determined, appropriate workspace constraints identified, and the strategy for workspace use is presented.

Chapter V introduces the stability model used throughout this thesis. *AquaRobot's* center-of-body and center-of-gravity relations are defined. The issue of static versus dynamic stability is addressed.

AquaRobot currently uses rectangular leg motions for all gaits. Curved leg motions result in smoother gaits. In Chapter VI, two smooth leg motion models are developed. Elliptical and cycloidal leg motion curves are generated for use in the *AquaRobot* simulator.

Statically stable tripod gait are planned and implemented in Chapter VII. To ease initial gait implementation, flat terrain, fixed body height, fixed body orientation, and

straight line path constraints are placed on the vehicle. Some constraints are lifted as the thesis progresses. Ultimately, a continuous omnidirectional tripod gait is developed.

Chapter VIII presents the *AquaRobot* simulation program. The simulation features and program organization are described. The final chapter summarizes the contributions of this thesis. Future research areas are also outlined. Appendix A contains an overview and history of the *AquaRobot*. Appendices B and C contain program code. Appendix D is an *AquaRobot* Simulator User's Manual that is separable from the thesis.

II. SURVEY OF PREVIOUS WORK

Three aspects of practical walking machine design are considered crucial: control of legged vehicles, actuator and leg design, and gait study. Vehicle control is considered the most crucial because of its inherent complexity. Many researchers are currently devoting extensive efforts to this research area. Only a few walking machines have sophisticated leg designs. Serious study has only recently been given to improving leg efficiency through better designs of legs and actuators. Finally, previous gait study has been concentrated on straight line motion over flat terrain. This thesis provides the foundation to extend gait study in *AquaRobot* beyond these motion and terrain constraints. [TOD85]

A A BRIEF HISTORY OF WALKING MACHINES

Walking machines have been constructed with from one to eight legs, and possibly more. An even number of legs is almost universal as this allows efficient gaits for progression in a straight line. More legs were typically used in the past for heavily loaded, but slower vehicles, while bipeds and quadrupeds were generally faster and more agile. Six legs is a magic number because it allows two alternating tripods. Two legs is another popular number because it allows emulation of human walking. Some of the many properties to consider when selecting the number of legs are: 1) stability, 2) energetic efficiency, 3) redundancy (using fewer legs to walk if some are unavailable), 4) joint control requirements, 5) cost, 6) weight, 7) desired complexity of sensing, and 8) possible gaits. [TOD85]

Two of the earliest examples of inventors' attempts to design walking machines are Rygg's mechanical horse in 1893 and Bechtolsheim's four-legged machine in 1913. There is no indication that either machine actually walked. [SON89]

In the mid 1950's, numerous research groups began to systematically study walking machines [SON89]. Then, in the 1960's, the Space General Corporation designed and constructed a six-legged machine and an eight-legged machine to investigate their applicability as lunar explorers. These two machines met their design goals, but exhibited poor terrain adaptability because of too few degrees of freedom. [TOD85]

In 1966, Frank and McGhee developed the first autonomous legged vehicle to walk under computer control. They called their walking machine the "Phoney Pony". This four-legged machine was powered externally via a cable and only walked in a straight line. In 1968, the General Electric Corporation built another quadruped, a 3000 pound vehicle with 12 degrees of freedom and a human rider-operator. The General Electric quadruped was successful at obstacle climbing and possessed good mobility in difficult terrain. Because its operation was highly complex and demanding, however, only a few people ever learned how to operate it. The General Electric quadruped proved the necessity for computer control of multi-degree-of-freedom vehicles. [TOD85]

In the 1970's, several walking machines were constructed. A pneumatically powered and analog computer controlled biped was successfully operated in Yugoslavia. A series of bipeds was developed in Japan, with walking speeds ranging from 90 seconds per step to the speed of a slow human walk. From 1974 through 1979, Russia developed and operated two six-legged walking machines of the insect body type. Finally, in 1977, McGhee and his associates at the Ohio State University (OSU) first got the OSU Hexapod to walk. [TOD85]

With the progress in microcomputer technology, computer-aided design, and computer simulation, the 1980's ushered in the development of several new walking machines. In 1980, Hirose and Umetani, at the Tokyo Institute of Technology, built the four-legged Perambulating Vehicle II (PVII). The PVII was promising because of its sophisticated pantograph leg design. Hirose, in 1984, constructed an enlarged version of the PVII, called TITAN III. In 1982, the Ohio State University developed monopod and six-legged walking machines as part of the Adaptive Suspension Vehicle (ASV) project. [TOD85]

More recently, Brooks, at the Massachusetts Institute of Technology, has designed and built several insectoid walking robots, one named Atilla, to demonstrate his theories on subsumption. Additionally, the Carnaegie-Mellon University is working on a walking machine, the Ambler, designed to explore the surface of Mars. Finally, research continues on *AquaRobot* at the Naval Postgraduate School and the Japanese PHRI.

B. SURVEY OF HEXAPOD VEHICLES

Many hexapod vehicles have been designed and built to advance research in walking control, leg and actuator design, and gait planning. Six legs is a good compromise between speed (shown in the next section), stability (generally improved with a larger number of legs), and simplicity [WAL84]. The incremental stability gain is smaller when going from six legs to eight legs than when going from four legs to six legs [WAL84]. So, although there is a marked advantage to using hexapod vehicles over quadruped vehicles, the increasing mechanical and computational complexity of eight-legged vehicles offsets their minimal stability advantages.

One of the first successful hexapod walking machines was that constructed at the University of Rome in 1972. This machine was simply a six-legged version of the Phoney

Pony. Then, as mentioned in the previous section, the OSU Hexapod walked in 1977. In 1983, Odetics, Incorporated constructed the ODEX I, a six-legged walking machine with some climbing ability. ODEX II, also a hexapod machine, has since been designed for use in nuclear power plants [ODE86]. In 1985, the design and construction of a large, man-carrying hexapod called the Adaptive Suspension Vehicle was completed at OSU [MCG86]. The ASV is a self-contained walking machine designed to traverse natural terrain. Each of the hexapods discussed has made a great contribution to the study of walking control, leg and actuator design, and gait planning. However, of all the known hexapods, only *AquaRobot* has moved beyond the research phase into practical use.

C. INTRODUCTION TO WALKING THEORY

The basic definitions and theorems for hexapod tripod gait planning used in this paper are introduced in this section. A *gait* is a method of locomotion distinguished by a specific pattern of lifting and placing of the feet. Gaits are often described using McGhee's and Jain's event sequence notation [MCG72]. In such a notation, the integer i corresponds to the event of placing foot i on the ground. Integer $i+n$, where n equals the number of legs, represents lifting the same foot. *AquaRobot's* legs are numbered sequentially in a clockwise direction from one through six.

The *transfer phase* of a leg is that time period when the foot is not touching the ground. The transfer phase is also known as the *recovery time* τ or the *return stroke* [TOD85]. The *support phase* t_s of a leg, then, is simply the opposite of the transfer phase: the time period when the foot is touching the ground. The *cycle time* T is the time required for a complete cycle of leg locomotion of a *periodic gait*. The cycle time includes the transfer and support phases of the leg. A *periodic gait* is one in which every limb operates

with the same cycle time. Otherwise, the gait is *non-periodic*. The duty factor β_i is the fraction of the cycle time in which leg *i* is in the support phase. [SON89] Thus,

$$\beta_i = \frac{time\ of\ support\ phase\ of\ leg\ i}{cycle\ time\ of\ leg\ i} = \frac{t_{si}}{T_i}. \tag{2.1}$$

A periodic gait is *singular* if there is a simultaneous occurrence of two or more events during a locomotion cycle. Conversely, a periodic gait is *nonsingular* if no two of its events occur simultaneously [MCG68a]. A gait is considered *regular* if all legs have the same duty factor, as when

$$\beta_i = \beta_j = \beta \qquad \begin{cases} i, j = 1,2,\ldots,n \\ n\ is\ the\ leg\ number \end{cases}. \tag{2.2}$$

The leg *stride* is a complete cycle of leg movements from a particular leg movement to the next occurrence of the same leg movement [TOD85]. The *stride frequency f* is the number of strides in unit time [TOD85]. The *stride length* λ is the distance the center of gravity of the body travels during one stride [TOD85]. The leg *stroke R* is the distance the foot travels, relative to the body, during the support phase [SON89]. A *step* is defined as the interval from the time a leg is placed until the time the next leg is placed [HIR84].

A *working volume* is associated with each leg. This volume is a subset of the three-dimensional space, defined relative to the body, consisting of the points which the foot can reach [KWA90]. The *support pattern* of a walking machine is defined as "the two dimensional point set in a horizontal plane consisting of the convex hull of the vertical projection of all foot points in the support phase" [SON89]. Generally, the *contact point* between the foot and the ground is idealized to have no slip. Further, although the actual foot contact may be distributed over some small surface area, the center of pressure may be chosen as the foot contact point. A *foothold* is described as a point on a piece of terrain

[KWA90]. When a foot is placed on the terrain, its foothold becomes the *support point* of the leg [KWA90]. A foothold can be assigned to a leg while it is still in the transfer phase.

There are four important definitions related to stability. First, *stability margin* S_m for an arbitrary support pattern, is the shortest distance from the vertical projection of the center of gravity to any point on the boundary of the support pattern in the horizontal plane [KWA90]. Then, the *front stability margin* and the *rear stability margin* describe distances from the vertical projection of the center of gravity to the forward and rearward boundaries of the support pattern, respectively [SON89]. The front and rear stability margins are measured along the direction of motion. Finally, the *longitudinal stability margin* S_l is defined as the shortest distance from the body's center of gravity to the boundary of the support pattern, measured in the direction of travel [MCG86]. From the definition of longitudinal stability margin, a gait is deemed *statically stable* if $S_l \geq 0$. Otherwise, it is deemed *statically unstable*.

With some fundamental gait definitions explained, we can now show that vehicle speed increases as the number of legs increases. Waldron et al. [WAL84] and Todd [TOD85] agree that the major limitation on speed of legged locomotion is the time required to return the leg through the air to its starting position. For any legged vehicle, there is a minimum time for a leg to be in this transfer phase. Additionally, there is a minimum duty factor for statically stable systems: $\beta_{quadruped}$=0.75 for quadrupeds, $\beta_{hexapod}$=0.5 for hexapods, and $\beta_{eight-legs}$=0.375 for eight legs. Finally, if the leg stroke R return stroke τ and cycle time T are presumed constant, the duty factor β causes the speed V to change as

$$V = \frac{R}{T\beta}. \tag{2.3}$$

From previous definitions,

$$T = t_{support\ phase} + t_{transfer\ phase} = t_s + \tau. \tag{2.4}$$

Then, using equation 2.1

$$t_s = T\left(\frac{t_s}{T}\right) = T\beta. \tag{2.5}$$

Combining equations 2.4 and 2.5

$$T = T\beta + \tau. \tag{2.6}$$

Then, for a fixed τ,

$$T = \frac{\tau}{1-\beta}. \tag{2.7}$$

Finally, combining equations 2.3 and 2.7 yields

$$V = \frac{R}{\tau}\left(\frac{1-\beta}{\beta}\right). \tag{2.8}$$

Substituting $\beta_{quadruped}$ into equation 2.8 yields

$$V = \frac{R}{3\tau}. \tag{2.9}$$

Using $\beta_{hexapod}$ in equation 2.8 yields

$$V = \frac{R}{\tau}. \tag{2.10}$$

Lastly, using $\beta_{eight-legs}$ in equation 2.8 provides

$$V = \frac{5R}{3\tau}. \tag{2.11}$$

From these relations, there is obviously a speed increase as the number of legs increases from four to eight. However, as previously noted, the speed increase from four legs to six legs is threefold, whereas the speed increase from six legs to eight legs is only

two-thirds. Herein lies one reason hexapod walking machines are so prevalent over other walking machines.

D. SUMMARY

This chapter provides background and introductory information helpful for understanding the gait planning research undertaken in this study. A general history of walking machines is followed with a more specific survey of hexapod walking machines. Terminology and methods typical of the gait planning problem are defined. Other, more specific, definitions are presented as material develops throughout the thesis.

The following chapter contains descriptions of *AquaRobot*, the model used to simulate *AquaRobot*, and the gait chosen for development in this thesis.

III. ROBOT DESCRIPTION AND SIMULATION MODEL

This chapter is intended as a description of the geometrical features of *AquaRobot*, a description of the *AquaRobot* model adopted for this study, and a discussion of the constraints employed to limit the scope of this study. Additionally, this chapter describes the specific gait planning and simulation issues undertaken in this thesis.

A. *AQUAROBOT* DESCRIPTION

As mentioned previously, *AquaRobot* is a six-legged, insect-type robot. The body is hexagonal and the legs are installed on the sides of the hexagonal frame. Figure 3-1 shows the major dimensions and sensor locations of *AquaRobot*. In this figure, the camera manipulator is shown positioned between legs one and two, with leg one pointing directly to the reader's right.

Figure 3-2 is a detailed plan view of *AquaRobot*. In this figure, the individual leg numbers are shown. The legs are placed at 60° increments about the body. This figure also includes a detail of the underwater TV camera, ultrasonic ranging device, and lights fitted to the manipulator boom mounted on top of the robot's body. The manipulator boom has three articulations similar to those found in the legs. The first articulation revolves about a vertical axis through the centerline of the robot's body. The next two articulations revolve about horizontal axes. Altogether, these three articulations give the camera three degrees of freedom.

Figure 3-3 presents a detailed side view of *AquaRobot*. Legs one and four are shown with dimensions representative of all six legs. The manipulator boom is shown fully extended with its maximum reach of 155 centimeters. This figure also shows leg

articulations representative of all six legs. The first articulation is located at the intersection of the legs and the body. This articulation rotates about a vertical axis and is also referred to as articulation one, joint one, the azimuth joint, or the *HIP*, depending on the reference. The typical HIP joint, then, is located 37.5 centimeters outboard from the center of the body at 0°, 60°, 120°, 180°, 240°, and 300° increments. The second articulation is 20 centimeters outboard of the HIP. It rotates about a horizontal axis and is also known as articulation two, joint two, the elevation joint, or *KNEE1*. The third articulation, also known as articulation three, joint three, the knee joint, or *KNEE2*, is 50 lineal centimeters outboard of KNEE1 and also rotates about a horizontal axis. The fourth, and final, articulation is known as articulation four or joint four. It is a passive ball-and-socket joint that attaches the foot to the leg and is 100 centimeters from KNEE2.

Figure 3-4 is a detailed view representing a typical leg's structure. In this figure, joint rotational configurations are shown. For the HIP, a positive angle represents a clockwise (CW) rotation of the joint. For the KNEE1 and KNEE2 joints, a positive angle represents lifting the leg segment. Although the foot is not actively (motor) controlled, it is allowed to rotate passively within ± 45 degrees about joint four. Figure 3-4 also shows cutaway views of the previously mentioned harmonic gears, bevel gears, DC motors, and motor encoders.

Table 3-1 provides a synopsis some of *AquaRobot's* more relevant characteristics. *AquaRobot*, with its present walking and control algorithms, can achieve a maximum speed of 7.75 meters/minute when walking on flat surface terrain using a *rectangular* alternating tripod gait. The rectangular terminology refers to the way the foot is moved: 1) first the foot is lifted directly upward, 2) then the foot is moved horizontally in the direction of travel for one stride, and 3) finally the foot is lowered directly to touch the

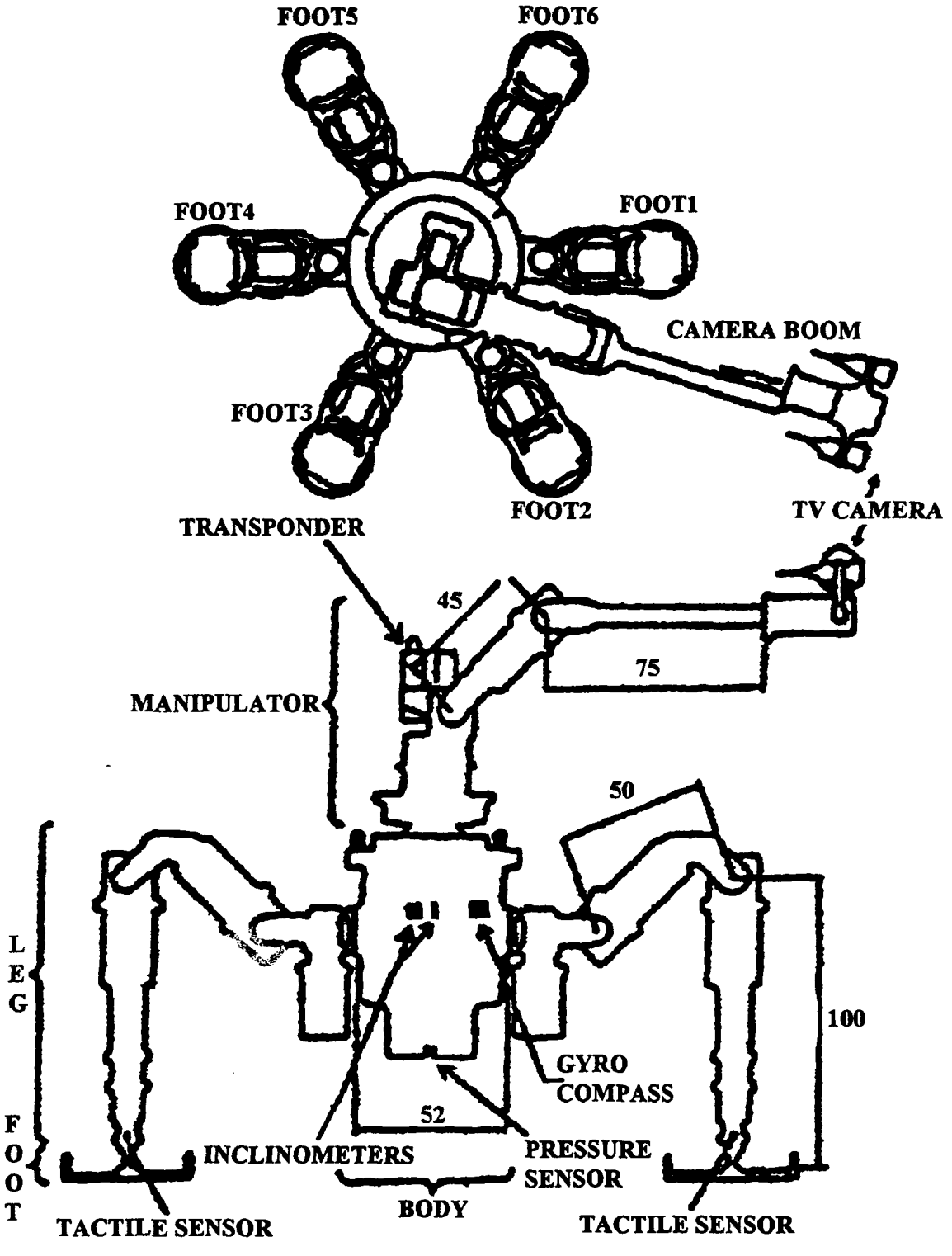


Figure 3-1. Major Dimensions and Sensors (After [IWA90]).

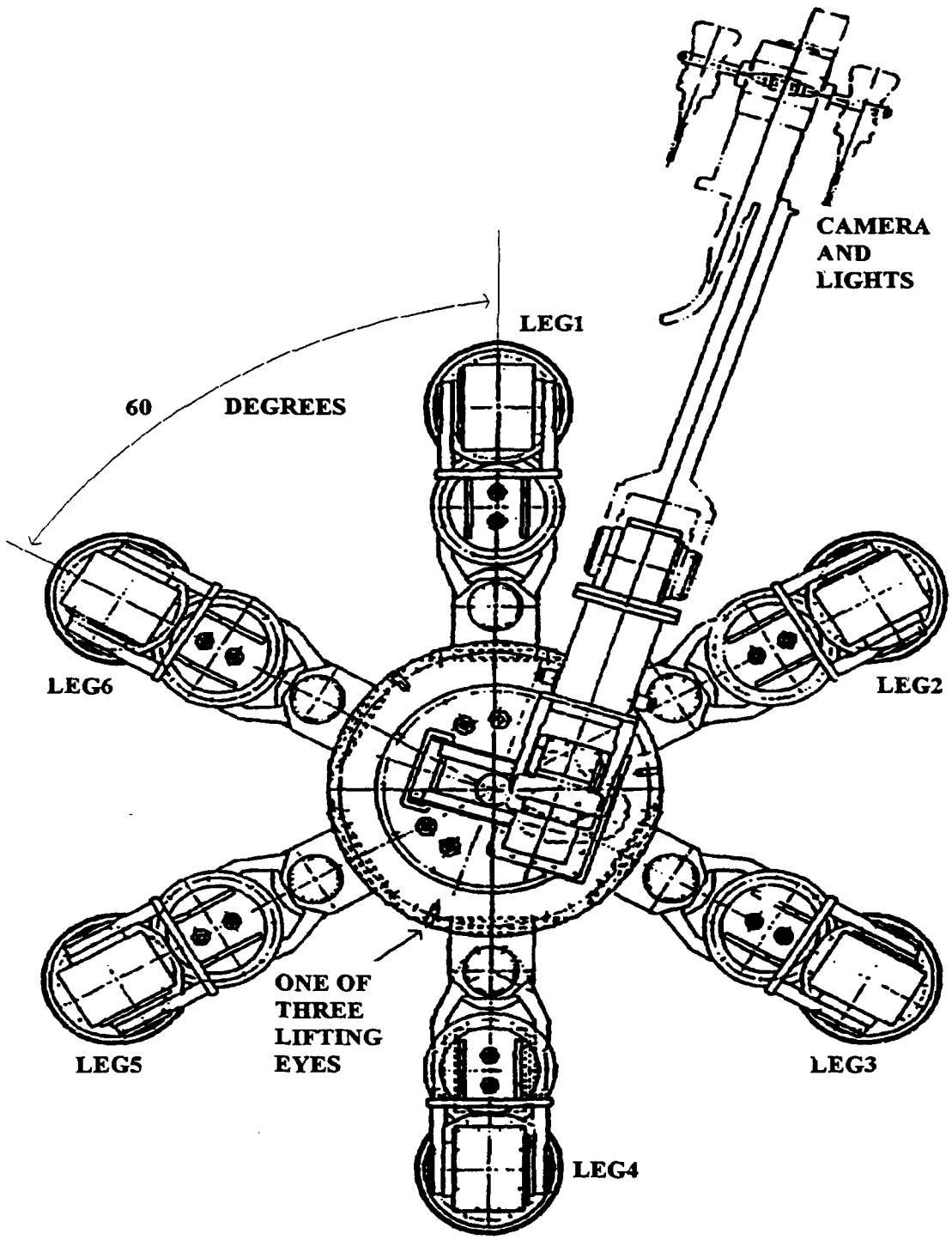


Figure 3-2. Plan View of *AquaRobot*.

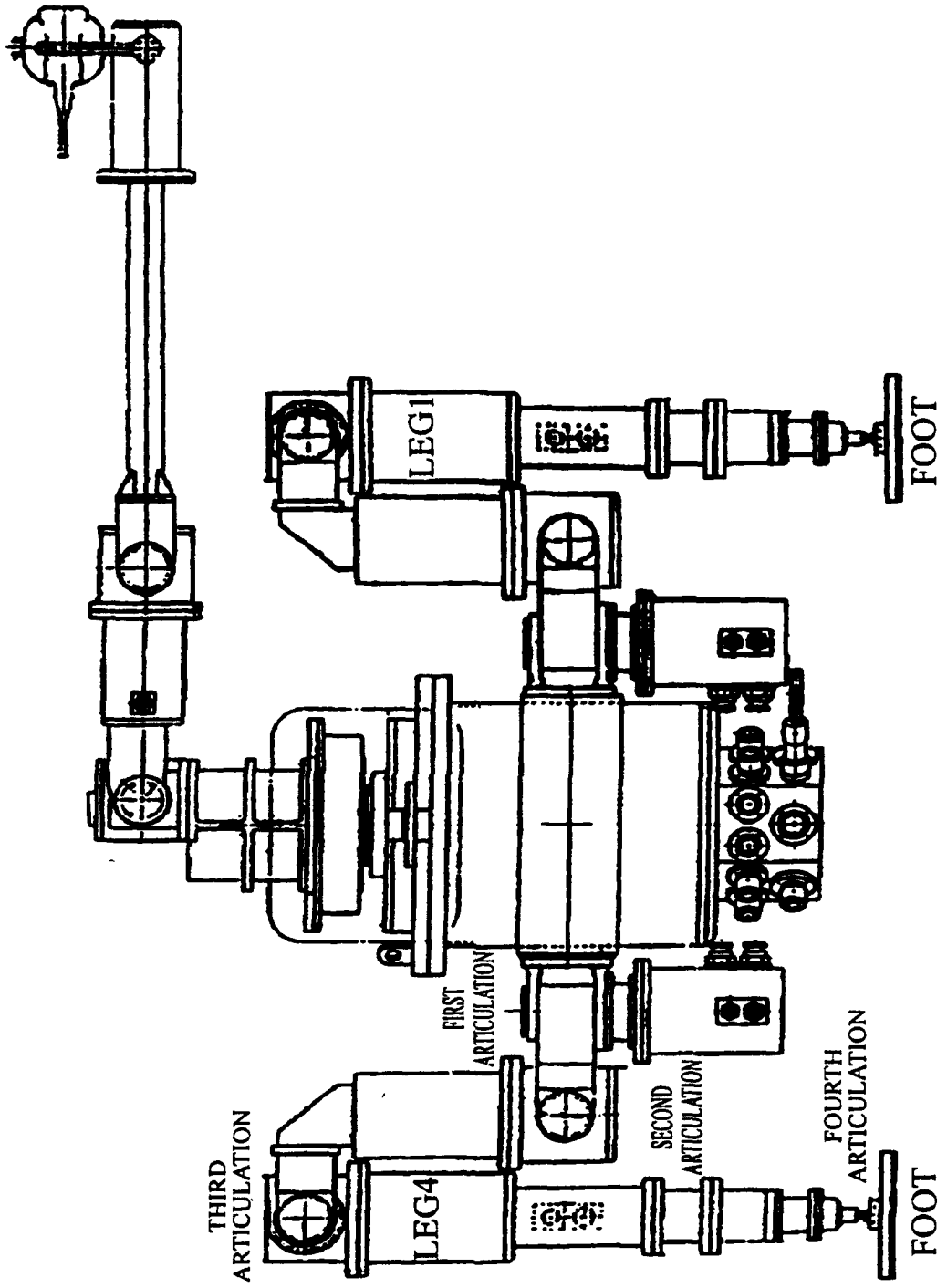


Figure 3-3. Side View of *AquaRobot*.

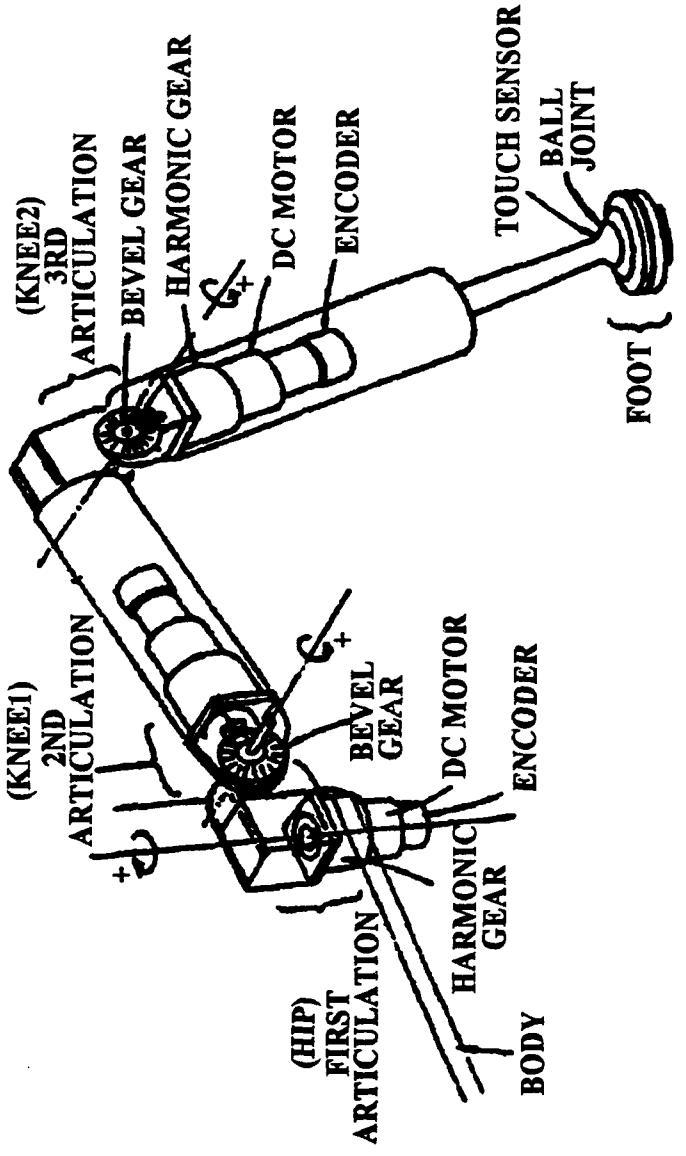


Figure 3-4. Leg Structure (After [AKI89]).

TABLE 3-1. *AQUAROBOT* PROTOTYPE CHARACTERISTICS (AFTER [IWA90]).

VEHICLE TYPE	axis-symmetric 6-legged insect-type walking
JOINT TYPE	4 joints per leg joint 1: active, vertical axis, revolute joint 2: active, horizontal axis, revolute joint 3: active, horizontal axis, revolute joint 4: passive, ball-and-socket
JOINT DRIVE METHOD	semi-direct drive, DC servo motor
CONTROL METHOD	80286-based microcomputer controlling hardware via software algorithms
MAXIMUM WALKING SPEED	7.75 meters/minute (flat land)
MAXIMUM ROTATING RATE	445 degrees/minute (flat land)
MAIN MATERIAL USED	anti-corrosive aluminum
VEHICLE WEIGHT	870 kgf (in air), 440 kgf (in water)
SENSORS	6 tactile sensors 2 inclinometers 1 flux gate gyrocompass 1 pressure sensor
MANIPULATOR CHARACTERISTICS	TV camera end-effector with ultrasonic range finding capability
MAXIMUM TERRAIN ROUGHNESS	± 35 centimeters
MINIMUM TERRAIN ROUGHNESS	± 5 centimeters
WATERTIGHT DEPTH	50 meters
TETHER CHARACTERISTICS	100 foot length 42 millimeter diameter 18 metal wire conductors optical fiber link 1500 kgf tensile strength
NAVIGATION	man-machine interface (XYZ inputs) dead reckoning transponder system
PURPOSES	measure flatness of rubble mound observe underwater structures supervise underwater construction

terrain. The foot *motion trace*, or imaginary path drawn in the air (or water) by the foot, would describe three sides of a rectangle.

AquaRobot can rotate on its vertical centerline in a CW or counter clockwise (CCW) direction if desired. The maximum rotating rate achieved is 445 degrees/minute. *AquaRobot* was designed to walk in terrain with a maximum roughness of ± 35 centimeters, representative of a typical unleveled rubble mound. A typical leveled rubble mound has a maximum roughness of ± 5 centimeters.

B. SIMULATION MODEL

The hexapod vehicle simulation model used in this thesis is the prototype *AquaRobot* presently under test at the Japanese PHRI. Figure 3-5 shows plan and side views of the simulation model of *AquaRobot* with the earth- and body-fixed coordinate systems. In this study, the earth-fixed coordinate system is known as the *world* (x_W, y_W, z_W) while the body-fixed coordinate system is known as the *body* (x_B, y_B, z_B). The body coordinate system is fixed at the center of the plane defined by the HIPs of all six legs. The x_B-axis lies along the line joining the HIPs of legs one and four and passing through the center of the body. Positive x is in the direction of leg one. The y_B-axis is perpendicular to the x_B-axis at the center of the body and passes between legs two and three and legs five and six. Positive y is in the direction between legs two and three. The z_B-axis lies orthogonal to the x_B and y_B axes with positive z following the right-hand rule convention: positive z_B is down.

Figure 3-6 shows the relationship between the physical and modeled walking machine. A representative side view that includes legs one and four is shown. Leg four (LEG4) is a scale drawing of an actual *AquaRobot* leg. Leg one (LEG1) is depicted in linkage form. Note from the figure that the joints are numbered from inboard outward,

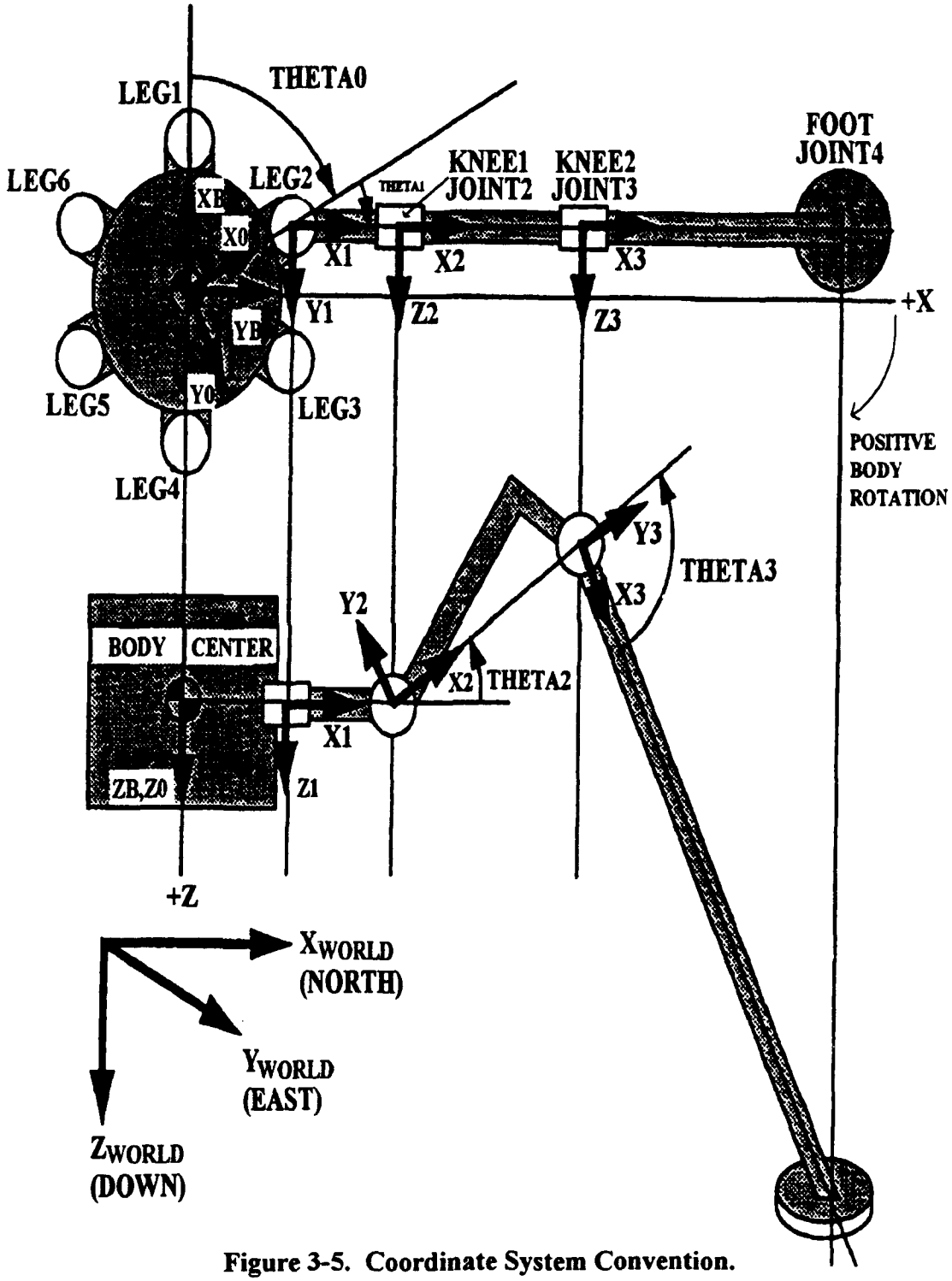


Figure 3-5. Coordinate System Convention.

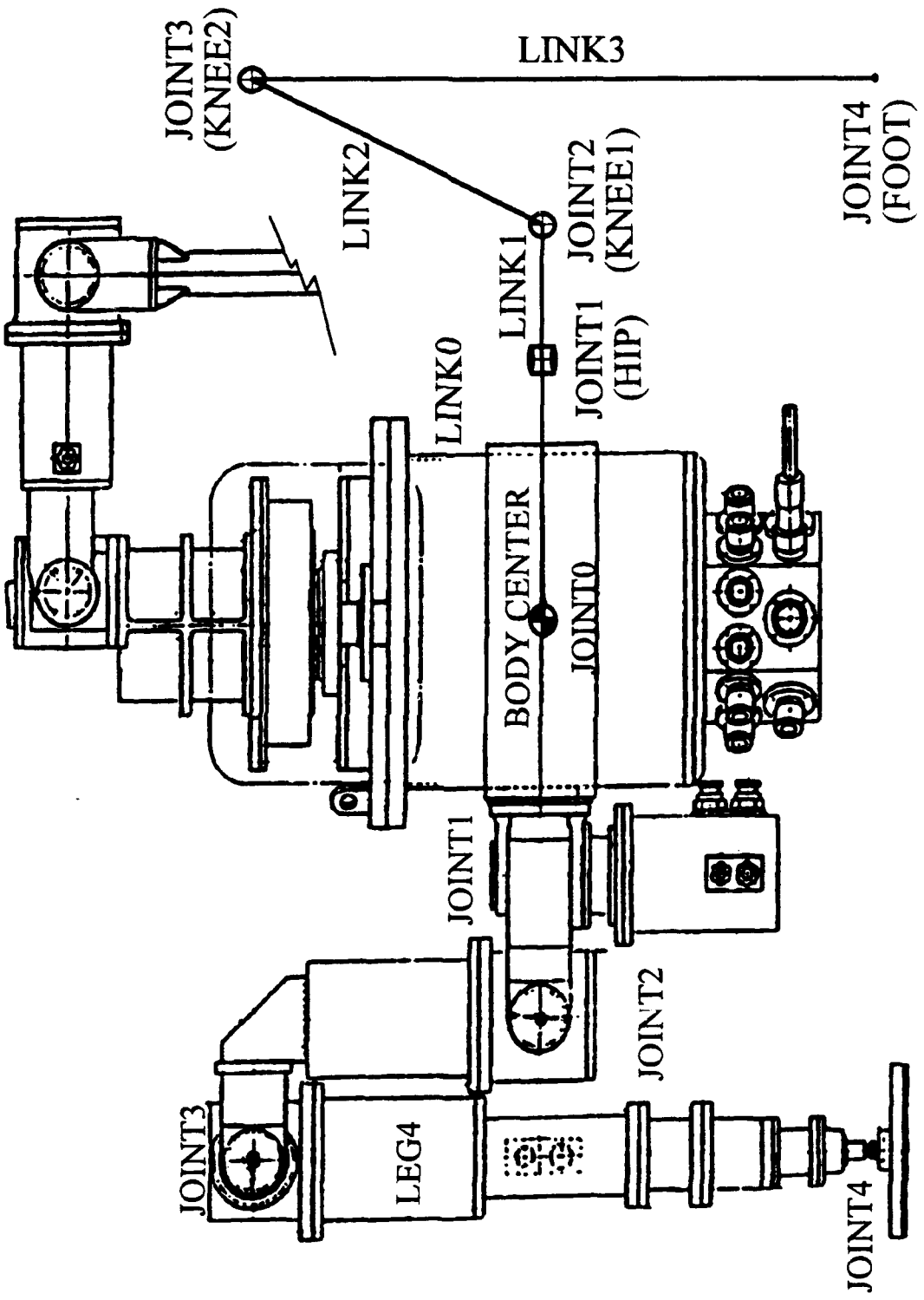


Figure 3-6. Relationship Between the Physical and Modeled Robot.

with joint zero (JOINT0) located at the robot's center-of-body. Links are numbered following the inboard joint. Therefore, link zero (LINK0) is between JOINT0 and joint one (JOINT1), link one (LINK1) is between JOINT1 and joint two (JOINT2), etc. The action of joint four (JOINT4) is not modeled in this study. Instead of modeling JOINT4 and the entire foot pad, a *point* foot is used where the junction of link three (LINK3) would meet JOINT4.

At the outset of this study, *AquaRobot* simulation system parameters were adopted to ensure come consistency in program development. The system is described in equations 3.1 through 3.6. Table 3-2 then lists the variables used in the system description and their meaning. The *System* is defined as a state vector that includes the states of the legs and body $\bar{x}$ and the warning flags $\bar{F}$. Then,

$$System = \left(\bar{x}, \bar{F}\right) \tag{3.1}$$

and

$$\bar{x} = \left(\bar{x}_{leg}, \bar{x}_{body}\right) = \left(\bar{x}_l, \bar{x}_b\right). \tag{3.2}$$

The leg states $\bar{x}_l$ are further subdivided into position, velocity, and acceleration vectors

$$\begin{aligned} \bar{x}_l &= \left(\beta_{k1}, \beta_{e1}, \beta_{a1}, \ldots, \beta_{k6}, \beta_{e6}, \beta_{a6}\right), \\ \dot{\bar{x}}_l &= \left(\dot{\beta}_{k1}, \dot{\beta}_{e1}, \dot{\beta}_{a1}, \ldots, \dot{\beta}_{k6}, \dot{\beta}_{e6}, \dot{\beta}_{a6}\right), \text{ and} \\ \ddot{\bar{x}}_l &= \left(\ddot{\beta}_{k1}, \ddot{\beta}_{e1}, \ddot{\beta}_{a1}, \ldots, \ddot{\beta}_{k6}, \ddot{\beta}_{e6}, \ddot{\beta}_{a6}\right). \end{aligned} \tag{3.3}$$

The body states $\bar{x}_b$ are then

$$\bar{x}_b = \left(x_E, y_E, z_E, \phi, \theta, \psi, u, v, w, p, q, r\right). \tag{3.4}$$

The flag vectors $\bar{F}$ provide warnings of foot contacts with the terrain and of joint limits reached. Therefore,

$$\bar{F} = \left(\bar{F}_{Contact}, \bar{F}_{Limit}\right) = \left(\bar{F}_C, \bar{F}_L\right). \tag{3.5}$$

The flag vectors are also subdivided into foot contact and joint limit sub-vectors

$$\begin{aligned} \bar{F}_C &= (C_1, \ldots, C_6), \text{ and} \\ \bar{F}_L &= (L_{k1}, L_{e1}, L_{a1}, \ldots, L_{k6}, L_{e6}, L_{a6}). \end{aligned} \tag{3.6}$$

These system state vectors allow code from several authors to work together without significant problems. Each author may still refer to parameters using terminology that best describes the use of that parameter. For example, this author refers to joint one as the HIP, joint two as KNEE1, and joint three as KNEE3.

TABLE 3-2. SIMULATION SYSTEM VARIABLE DESCRIPTIONS.

β	Joint Angular Displacement
$k_1, \ldots, k_6$	Knee joint (KNEE2), legs 1 through 6
$e_1, \ldots, e_6$	Elevation joint (KNEE1), legs 1 through 6
$a_1, \ldots, a_6$	Azimuth joint (HIP), 1 through 6
x_E or x_W	North
y_E or y_W	East
z_E or z_W	Down
ϕ	Roll
θ	Elevation
ψ	Azimuth
u	Body velocity with respect to earth in body coordinates (principal axes).
v	
w	
p	Body angular rate in body coordinates.
q	
r	
$C_1, \ldots, C_6$	Foot contact flag, legs 1 through 6
L	Joint limit flag, all 18 joints

Many simplifying assumptions were made within this model of *AquaRobot*. The simplifications were made primarily to speed development of the gait planning algorithms and graphics routines. However, each program was devised with future expansion in mind.

Whenever possible, program code was modularized for future expansion. Specific simplifications are addressed in the following paragraphs.

1. Flat Terrain

The first simplifications concern the vehicle's operating environment. Although the actual *AquaRobot* was designed to operate in uneven terrain, the simulation model initially developed included only flat terrain with no obstacles. The flat terrain is graphically displayed using a checkerboard design. The simulation process does allow feedback of foot contacts with the flat terrain. Expansion to allow foot contact feedback when the foot encounters unstructured terrain is present.

2. Fixed Body Height

Because the terrain is flat, the robot's body height and horizontal body angle are fixed arbitrarily at the outset of the program. *AquaRobot* has both selectable body height and terrain following capability. Neither of these features is necessary to implement a tripod gait. Selectable body height is a feature of the simulation. Terrain following capability can easily be added as a code module if future research warrants its use.

3. Straight Line Path

Again, for ease in initial program development, a straight line path oriented with leg one following the positive x_W-axis was adopted. When turning is considered, straight line paths between two points is still used. However, the robot is not required to rotate to have LEG1 align with the desired direction of travel. Some smooth path-following techniques were implemented as part of this thesis.

4. Constrained Workspace

Because of its leg design, *AquaRobot* has the ability to step into or on its own feet or legs. Although this feature allows much greater freedom when determining footholds, it is not beneficial at the outset of program design. Therefore, the workspace of *AquaRobot* was constrained so no leg or foot overlap was allowed. Other workspace constraints, or none at all, are possible by modifying the existing program.

5. Dynamics

Finally, the effects of dynamics and hydrodynamics on gait operation are not covered. This model encompasses only the kinematic features of *AquaRobot*. This means the model imposes no limits on vehicle acceleration. Similarly, although the manipulator boom, tether, lifting lines, and transponder systems were cursorily described, they are not modeled nor considered in this thesis.

C. PROBLEM STATEMENT

This thesis concentrates on designing an improved alternating tripod gait for *AquaRobot*. Simulation of the improved design is an inherent design goal. Other specific design goals include:

- smooth leg motion,
- omnidirectional body movement,
- continuous body motion, and
- limited path following capability.

The laboratory director from Japan's PHRI agreed that implementing these improvements in an alternating tripod gait design would provide immediate increased effectiveness over the existing design [TAK93].

The selection of a gait is a very difficult problem. McGhee [MCG68a] showed that 39,916,800 possible nonsingular periodic hexapod gaits exist. Further, McGhee [MCG85] suggests the total number of hexapod gaits is actually a larger, unknown number. This thesis, however, is restricted to a single type of gait sequence known as the tripod gait. The tripod gait is a singular gait because three legs (hence, the term tripod) are placed simultaneously. The gait alternates between sets of three legs. LEGs (1, 3, 5) are considered tripod zero (TRIPOD0) and LEGs (2, 4, 6) are considered tripod one (TRIPOD1). This zero and one terminology is in keeping with C and C++ programming conventions.

The *alternating tripod gait* described above is particularly important for walking machines, and is another reason for the prevalence of hexapod vehicles [TOD85]. Six legs always allows a supporting tripod, even when half the legs are in the transfer phase. Therefore, it allows reasonable walking speed while always maintaining static stability.

D. SUMMARY

This chapter outlined the geometrical features, physical constraints, and model simplifications of the *AquaRobot* underwater walking robot. Next, the scope of the work this thesis undertakes was described. The following chapter contains the development and implementation of the kinematic and kinematics-related concerns of the *AquaRobot* when applied to an alternating tripod gait design.

IV. KINEMATICS

In this chapter, we consider the kinematics problem of computing the position and orientation of *AquaRobot's* foot, relative to the center of the body, given the joint angles for the HIP, KNEE1, and KNEE2. Next, we investigate the more difficult inverse kinematics problem: given the desired position and orientation of the foot relative to the body center, compute the set of joint angles which will achieve that result. Then a method of transforming between the world and body coordinate systems is derived. Next, the concept of workspace and its effects on walking is investigated. Kinematics, inverse kinematics, coordinate transformation, and workspace discussions are all developed using methods from Craig [CRA86]. The chapter ends with a discussion of the various body postures used in this thesis.

A. KINEMATICS

A typical manipulator, in this case a leg for *AquaRobot*, is actually a set of bodies connected in a chain by joints. The bodies are called *links*. For the purposes of kinematics, links are considered only as rigid bodies which define the relationship between two neighboring joint axes. Joints, then, form the connection between neighboring links. Joints are typically classed as either *revolute* or *prismatic*. Revolution about some axis describes a revolute joint. Sliding action along some axis describes a prismatic joint. All of *AquaRobot's* joints are revolute. [CRA86]

At the end of the chain of links that make up the manipulator is the *end-effector*. In our case, the end-effector is the foot at the end of the leg. Specifically, an end-effector for *AquaRobot* is a point foot at the end of LINK3 on each leg. The position of the foot with

respect to the center of the body is described by defining a foot {FOOT} *frame* relative to the center-of-body {CB} *frame*. A frame is the set of four vectors giving position and orientation information. Three of the vectors are typically unit vectors defining the principal axes of the frame. The other vector locates the origin of the frame with respect to some other frame, including the origin of the *world* {W} frame. [CRA86]

Four quantities are required to kinematically describe a manipulator. Of these, two describe the link and two describe the link's connection between adjoining links. For revolute joints, θ_i is known as the *joint variable* and the other three quantities are known as *link parameters*. Mechanisms defined with these conventions are typically described using *Denavit-Hartenberg* (DH) notation [DEN55]. There are many related conventions that ascribe to the name Denavit-Hartenberg. In this thesis, frame $\{i\}$ is attached to link i and has its origin lying on joint axis i. In other words, link i follows inboard joint i. [CRA86]

Figure 4-1 is a kinematic detail of a typical leg from the center-of-body to the foot. Leg frames are defined according to their respective joints. The {CB} frame corresponds to an imaginary frame zero {0}. Frame {0} does not move and is considered the reference frame for *AquaRobot*. Frame {1} is not actually a frame; however, for ease of performing the kinematics, each leg is treated as if it were a rotation of 60° about JOINT0. Therefore, LEG1 is at 0°, LEG2 is at 60°, LEG3 is at 120°, etc. With this in mind, Figure 4-1 shows the progression of frames from inboard outward as {CB}, {0}, {HIP}, {KNEE1}, {KNEE2}, and {FOOT}. Recall that JOINT4 and the entire foot is not used.

The leg itself is referenced to two other frames. First is the *center-of-gravity* {CG} frame. This frame locates the center of gravity of *AquaRobot*, which is not necessarily the same point as the center of body. For our purposes, {CG} is set equal to {CB}. Then

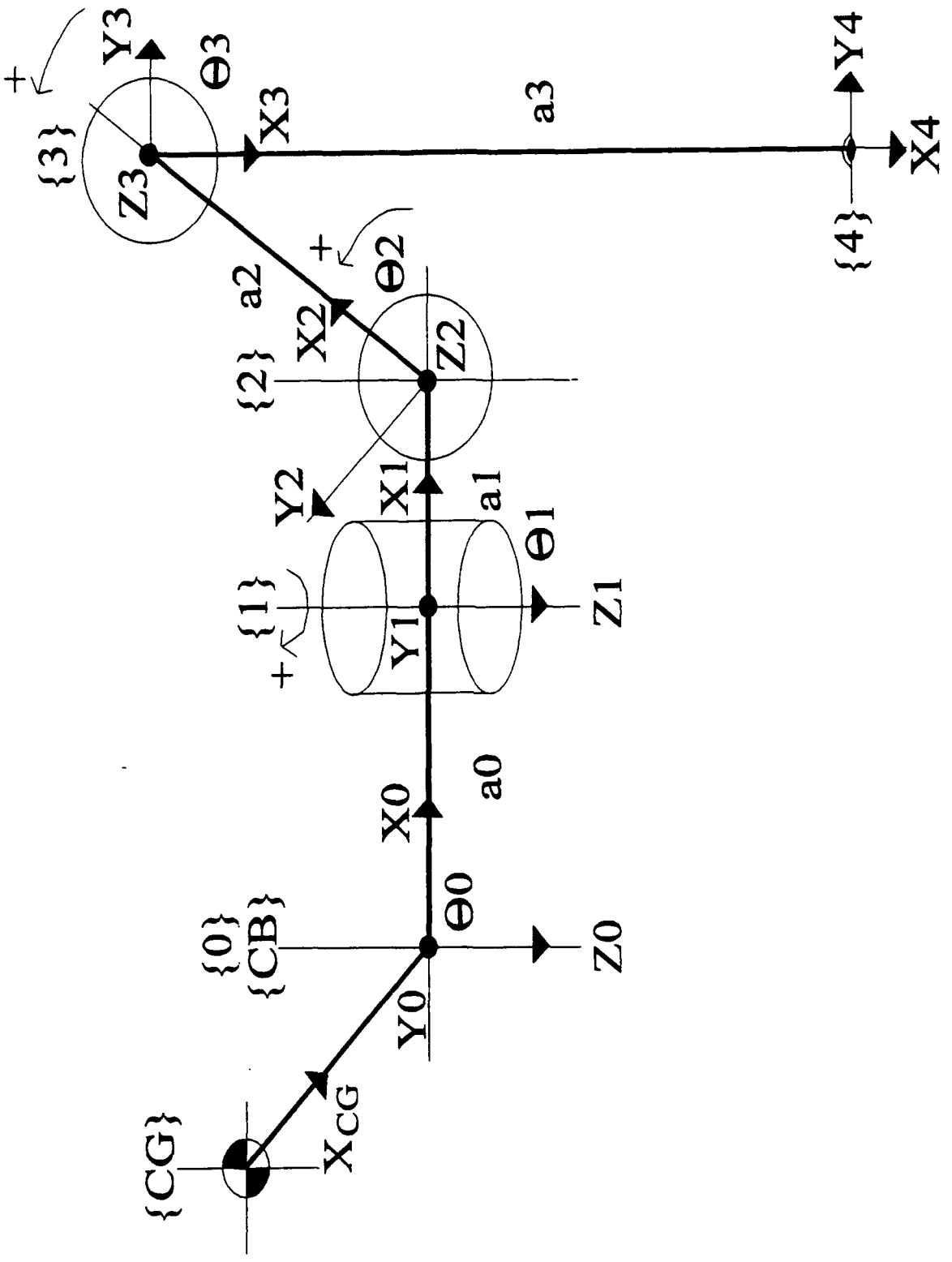


Figure 4-1. Kinematic Detail of Leg.

{CG} is referenced to {W}. The {CG} and {W} frames are shown only for perspective and are not used in the derivation of leg kinematics. Transformations between {W} and {CB}, however, are examined later in this chapter.

The general form of the transformation that relates the frames attached to neighboring links using the joint variable and link parameters previously defined is

$$ {}^{i-1}_{i}T = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i \cos\alpha_{i-1} & \cos\theta_i \cos\alpha_{i-1} & -\sin\alpha_{i-1} & -\sin\alpha_{i-1} d_i \\ \sin\theta_i \sin\alpha_{i-1} & \cos\theta_i \sin\alpha_{i-1} & \cos\alpha_{i-1} & \cos\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.1} $$

where

α_{i-1} is the angle between Z_{i-1} and Z_i measured positive about X_{i-1},
a_{i-1} is the distance from Z_{i-1} to Z_i measured along X_{i-1},
d_i is the distance from X_{i-1} to X_i measured along Z_i, and
θ_i is the angle between X_{i-1} and X_i measured about Z_i.

Table 4-1 lists the link parameters for a representative leg. Once the link frames are defined and the corresponding link parameters found, values from Table 4-1 are used in equation 4.1 to compute the transformations for each link. The {CB} to {0} transformation results in

$$ {}^{CB}_{0}T = \begin{bmatrix} C_0 & -S_0 & 0 & 0 \\ S_0 & C_0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.2} $$

where C_0 represents $\cos\theta_0$ and S_0 represents $\sin\theta_0$. Note that the distance a_{-1} between {CB} and {0} is zero centimeters because these two frames overlap. The {0} to {1} transformation results in

$$ {}^{0}_{1}T = \begin{bmatrix} C_1 & -S_1 & 0 & a_0 \\ S_1 & C_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.3} $$

where a_0 is the distance between {0} and {1}, 37.5 centimeters.

TABLE 4-1. LINK PARAMETERS FOR *AQUAROBOT*.

i	α_{i-1} (degrees)	a_{i-1} (cm)	d_i (cm)	θ_i (cm)	Range
-1	α_{WORLD}	a_{WORLD}	d_{WORLD}	θ_{WORLD}	
0	0	0	0	θ_0	$\theta_0 = 0°, 60°, 120°, 180°, 240°, 300°$
1	0	a_0	0	θ_1	$a_0 = 37.5$, $-60° \le \theta_1 \le 60°$
2	-90	a_1	0	θ_2	$a_1 = 20.0$, $-106.6° \le \theta_2 \le 73.4°$
3	0	a_2	0	θ_3	$a_2 = 50.0$, $-156.4° \le \theta_3 \le 23.6°$
4	0	a_3	0	θ_4	$a_3 = 100.0$, $-45° \le \theta_4 \le 45°$

The other transformations follow the same pattern:

$$ {}^{1}_{2}T = \begin{bmatrix} C_2 & -S_2 & 0 & a_1 \\ 0 & 0 & 1 & 0 \\ -S_2 & -C_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.4} $$

$$ {}^{2}_{3}T = \begin{bmatrix} C_3 & -S_3 & 0 & a_2 \\ S_3 & C_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.5} $$

and $$ {}^{3}_{4}T = \begin{bmatrix} C_4 & -S_4 & 0 & a_3 \\ S_4 & C_4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.6} $$

Because the entire foot is not used, we are not concerned with the action of JOINT4. Therefore, setting $\theta_4 = 0$ produces a substitution for equation 4.6:

$$ {}^{3}_{4}T = \begin{bmatrix} 1 & 0 & 0 & a_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.7} $$

which provides the necessary transformation from KNEE2 to the FOOT. The final matrix used to determine the transformation from {CB} to {4} ${}^{CB}_{4}T$ is found by multiplying each of the individual transformation matrices together:

$$ {}^{CB}_{4}T = {}^{CB}_{0}T\ {}^{0}_{1}T\ {}^{1}_{2}T\ {}^{2}_{3}T\ {}^{3}_{4}T. \tag{4.8} $$

The derivation of the final transformation matrix, including intermediate results, is shown in equations 4.9 through 4.12.

$$ {}^{CB}_{1}T = {}^{CB}_{0}T\ {}^{0}_{1}T = \begin{bmatrix} C_{01} & -C_{01} & 0 & a_0C_0 \\ S_{01} & S_{01} & 0 & a_0S_0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.9} $$

$$ {}^{CB}_{2}T = {}^{CB}_{0}T\ {}^{0}_{1}T\ {}^{1}_{2}T = \begin{bmatrix} C_{01}C_2 & -C_{01}S_2 & -S_{01} & a_0C_0 + a_1C_{01} \\ S_{01}C_2 & -S_{01}S_2 & C_{01} & a_0S_0 + a_1S_{01} \\ -S_2 & -C_2 & 0 & -a_2S_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.10} $$

$$ {}^{CB}_{3}T = {}^{CB}_{0}T\,{}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T = \begin{bmatrix} C_{01}C_{23} & -C_{01}S_{23} & -S_{01} & a_0C_0 + C_{01}(a_1 + a_2C_2) \\ S_{01}C_{23} & -S_{01}S_{23} & C_{01} & a_0S_0 + S_{01}(a_1 + a_2C_2) \\ -S_{23} & -C_{23} & 0 & -a_2S_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.11} $$

and $$ {}^{CB}_{4}T = {}^{CB}_{0}T\,{}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T\,{}^{3}_{4}T $$

$$ = \begin{bmatrix} C_{01}C_{23} & -C_{01}S_{23} & -S_{01} & a_3C_{01}C_{23} + a_0C_0 + C_{01}(a_1 + a_2C_2) \\ S_{01}C_{23} & -S_{01}S_{23} & C_{01} & a_3S_{01}C_{23} + a_0S_0 + S_{01}(a_1 + a_2C_2) \\ -S_{23} & -C_{23} & 0 & -a_3S_{23} - a_2S_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.12} $$

where

$C_i = \cos\theta_i$,
$S_i = \sin\theta_i$,
$C_{ij} = C_iC_j - S_iS_j = \cos(\theta_i + \theta_j)$,
and $S_{ij} = S_iC_j - C_iS_j = \sin(\theta_i + \theta_j)$.

Two Matlab programs were written to verify the derivation of the kinematics transformations. Program jcoord.m uses equation 4.12 to compute the (x, y, z) coordinates of the HIP, KNEE1, KNEE2, and FOOT from joint angles the user inputs. The Matlab program kin.m is a function that provides a similar output to the c3.m program when called. These two code modules are included in Appendix B.

B. INVERSE KINEMATICS

In the first section of this chapter, we focused on the *direct kinematics* of the leg. That is, we computed the position and orientation of {4} with respect to {CB}. In this section, we focus on the *inverse kinematics* of the leg. Our goal here is to find the required set of joint angles $(\theta_0, \theta_1, \theta_2, \theta_3)$ to place the foot, relative to the center of the body, when the desired foot position and orientation are known. There are two

approaches to solving the inverse kinematics problem. The *algebraic* approach is basically one of manipulating the link parameters into kinematic equations of a form for which a solution is known. The *geometric* approach is to try to decompose the spatial geometry of the leg into several lesser plane geometry problems. The geometric approach is the method used here. Figure 4-2 shows the plane geometry associated with *AquaRobot*. [CRA86]

First, consider the triangle formed by the CB, HIP, and FOOT of representative LEG2 and shown in the plan view of Figure 4-2. We can apply the Law of Cosines to solve for θ_1:

$$p_x^2 + p_y^2 = a_0^2 + b_1^2 - 2a_0 b_1 \cos(180° - \theta_1) = a_0^2 + b_1^2 + 2a_0 b_1 \cos\theta_1. \qquad (4.13)$$

The foot position is given in body-fixed coordinates $(p_{x_B}, p_{y_B}, p_{z_B})$. Then, we have

$$\cos\theta_1 = C_1 = \frac{p_x^2 + p_y^2 - a_0^2 - b_1^2}{2a_0 b_1}, \qquad (4.14)$$

where

$$b_1 = \sqrt{(p_x - a_0\cos\theta_0)^2 + (p_y - a_0\sin\theta_0)^2}. \qquad (4.15)$$

Finally, we use the two-argument arctangent (Atan2) to find

$$\theta_1 = \text{Atan2}\left(\pm\sqrt{1 - C_1^2}, C_1\right). \qquad (4.16)$$

Now we consider the triangle formed by the KNEE1, KNEE2, and FOOT joints as shown in the cross-sectional view of Figure 4-2. First, we find expressions for angles β and ψ, where

$$\psi = \theta_2 + \beta. \qquad (4.17)$$

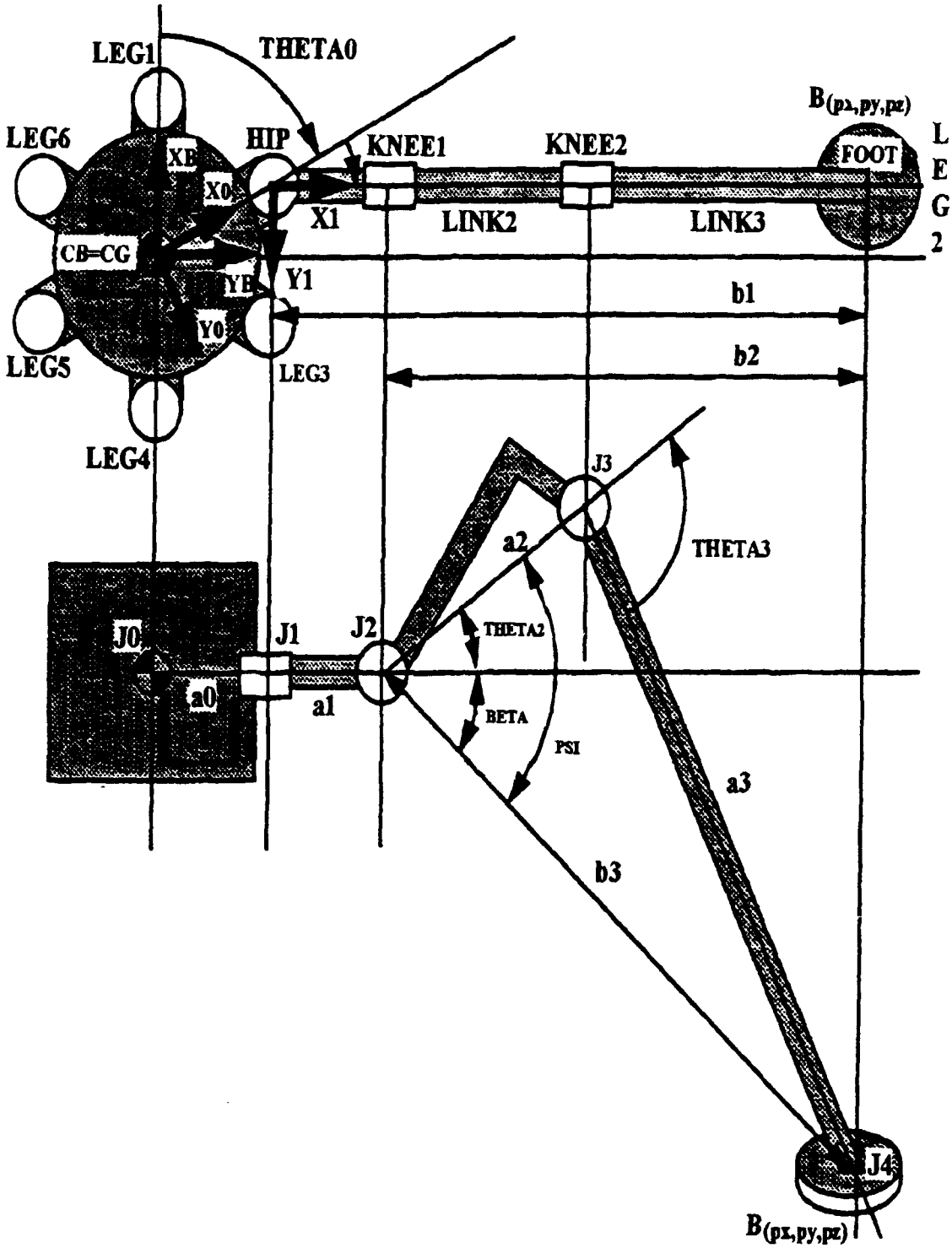


Figure 4-2. Plane Geometry of *AquaRobot*.

Then,

$$\tan\beta = \frac{p_z}{b_2}, \tag{4.18}$$

and

$$\beta = \text{Atan2}(p_z, b_2), \tag{4.19}$$

where

$$b_2 = b_1 - a_1. \tag{4.20}$$

Applying the Law of Cosines again to find ψ:

$$a_3^2 = a_2^2 + b_3^2 - 2a_2 b \cos\psi. \tag{4.21}$$

Then, we have

$$\cos\psi = \frac{a_2^2 + b_3^2 - a_3^2}{2a_2 b_3} \tag{4.22}$$

and

$$\sin\psi = \pm\sqrt{1-\cos^2\psi}, \tag{4.23}$$

where

$$b_3 = \sqrt{b_2^2 + p_z^2}. \tag{4.24}$$

Now, substituting equations 4.18, 4.22, and 4.23 into equation 4.17 yields

$$\theta_2 = \psi - \beta \tag{4.25}$$

and

$$\theta_2 = \text{Atan2}\left(\pm\sqrt{1-\cos^2\psi}, \cos\psi\right) - \text{Atan2}(p_z, b_2). \tag{4.26}$$

Finally, we again apply the Law of Cosines to the triangle formed by the KNEE1, KNEE2, and FOOT joints to solve for θ_3:

$$b_3^2 = a_2^2 + a_3^2 - 2a_2a_3\cos(180° - \theta_3) = a_2^2 + a_3^2 + 2a_2a_3\cos\theta_3. \tag{4.27}$$

We then have

$$\cos\theta_3 = C_3 = \frac{b_3^2 - a_2^2 - a_3^2}{2a_2a_3} \tag{4.28}$$

and

$$\theta_3 = \text{Atan2}\left(\pm\sqrt{1 - C_3^2}, C_3\right). \tag{4.29}$$

Because equations 4.16, 4.26, and 4.29 include the possibility of a positive or negative result (±), there are eight potential solutions for a single goal. Physical joint limitations may prevent using some of the potential solutions. Of the remaining valid solutions, the one closest to the present leg configuration is generally chosen. Still, all chosen solutions are tested using the direct kinematics routines previously developed. If the resulting kinematics solution yields a foot position equal to the desired foot position, we accept the inverse kinematics solution as the real solution. We have solved for θ_1, θ_2, and θ_3. The solution for θ_0 is already known because it is the angle that determines which leg is selected. For example, if we know we are trying to place FOOT2, then we know θ_0 is at 60°. This is because the HIP joints are located at constant, known angles about the body.

C. COORDINATE TRANSFORMATIONS

The *AquaRobot* simulation uses two coordinate systems in its calculations: the world (x_W, y_W, z_W) coordinate system and the body (x_B, y_B, z_B) coordinate system. The world coordinate system is used when it is necessary to specify absolute positions or velocities of

the body center, feet, or terrain. The world coordinate system is defined with the z_W-axis positive downward, with the x_W-axis positive when pointing North, and the y_W-axis positive when pointing East.

The body coordinate system, composed of three dimensional local coordinates fixed to the robot's body, is used to describe the coordination between the body and the legs and to determine stability. The origin of the body coordinate system is defined as the center of the plane defined by the HIPs of all six legs. The positive x_B-axis lies along the line joining the CB and the HIP of LEG1. The positive y_B-axis is perpendicular to the x_B-axis at the center of the body and passes between LEG2 and LEG3. The z_B-axis lies orthogonal to the x_B- and y_B-axes with positive z_B pointing downward.

The method of transforming between the body coordinate system and the world coordinate system is defined through the use of a 4 x 4 homogeneous transformation matrix, ${}^W_BT$ [CRA86]. The matrix may be partitioned into sub matrices

$$ {}^W_BT = \begin{bmatrix} {}^W_BR & {}^WP_B \\ 0 & 1 \end{bmatrix}, \tag{4.30} $$

where ${}^W_BR$ is a 3 x 3 rotation matrix and ${}^WP_B$ is a 3 x 1 position vector [LEE88a]. The vector ${}^WP_B$ represents the position of the body in the world coordinate system and ${}^W_BR$ gives the rotation of the body relative to the world coordinate system. The ${}^W_BR$ rotation matrix is derived using the *X-Y-Z fixed angle* orientation convention [CRA86]. *Fixed angles* mean the rotations are specified about a non-moving reference system. For our purposes, rotations about the x-axis are known as *roll* (ϕ), rotations about the y-axis are known as *pitch* or *elevation* (θ), and rotations about the z-axis are known as *yaw* or *azimuth* (ψ). The rotation matrix ${}^W_BT$ then becomes

$$ {}^{W}_{B}R_{XYZ(\phi,\theta,\psi)} = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix}. \tag{4.31} $$

The ${}^{W}_{B}T$ matrix allows transformation from body coordinates to world coordinates using the relationship

$$ \begin{bmatrix} x_W \\ y_W \\ z_W \\ 1 \end{bmatrix} = {}^{W}_{B}T \begin{bmatrix} x_B \\ y_B \\ z_B \\ 1 \end{bmatrix}, \tag{4.32} $$

where the position vectors $[x_W \quad y_W \quad z_W \quad 1]^T$ and $[x_B \quad y_B \quad z_B \quad 1]^T$ describe the same point in space in world and body coordinates, respectively. The Matlab code module b2wtrans.m, found in Appendix B, performs the body-to-world transformation.

To convert from world coordinates to body coordinates requires a similar transformation. In this case, the inverse of the ${}^{W}_{B}T$ matrix is used. Then, using the form of equation 4.32, we find the new transformation process is

$$ \begin{bmatrix} x_B \\ y_B \\ z_B \\ 1 \end{bmatrix} = {}^{W}_{B}T^{-1} \begin{bmatrix} x_W \\ y_W \\ z_W \\ 1 \end{bmatrix}. \tag{4.33} $$

The Matlab code module w2btrans.m, found in Appendix B, performs the world-to-body coordinate transformation.

D. WORKSPACE

In this section, *AquaRobot's workspace* is computed. Workspace is that volume of space which an end-effector, in our case, a foot, can reach [CRA86]. The workspace is also known as the *working volume* The mechanical limits of the joints restrict leg motion

and are a major factor to consider when developing control algorithms for a legged vehicle [LEE88a]. Because each leg of *AquaRobot* has the same geometrical configuration and joint limits, the working volumes of the legs are identical. The limits of the joint variables for a representative LEG are shown in Table 4-2.

TABLE 4-2. REPRESENTATIVE LEG JOINT LIMITS.

LEG Angles (LEG1 - LEG6)	$\theta_0 = 0°, 60°, 120°, 180°, 240°, 300°$
HIP Joint Limits	$-60° \le \theta_1 \le 60°$
KNEE1 Joint Limits	$-106.6° \le \theta_2 \le 73.4°$
KNEE2 Joint Limits	$-156.4° \le \theta_3 \le 23.6°$
FOOT Joint Limits	$-45° \le \theta_4 \le 45°$

These joint variable limits, then, separate the *reachable area* from the *unreachable area* [MCG79]. Reachable areas move with the body. The region included within the reachable area is known as the *unconstrained working volume* (UWV).

The *constrained working volume* (CWV) is defined as a subset of the original working volume, for each leg, that ensures static stability. Therefore, the CWV sets soft limits for each leg so as to exclude points from the working volume that may lead to instability [LEE88b]. In our case, the working volume is also constrained to prevent leg collisions. An *excluded area* for *AquaRobot's* legs, then, is that part of the reachable area where, if a foot were placed there, instability or leg collision might result [MCG79].

1. Unconstrained Workspace

The unconstrained horizontal workspace of *AquaRobot* is shown in Figure 4-3. The reachable areas include the sections in the *xy* plane around the individual HIPs and

within the mechanical joint limits. The unconstrained vertical workspace, or Z-plane reachable area, depends on the height of the vehicle's center-of-body above the terrain and is not shown in Figure 4-3.

To define the unconstrained vertical workspace, we must define the maximum and minimum reach of *AquaRobot*. If a leg were extended to its fullest, the added lengths of LINK0 (37.5 cm), LINK1 (20.0 cm), LINK2 (50.0 cm), and LINK3 (100.0 cm) would equal 207.5 centimeters. This length does not include the diameter (25.0 cm) or the thickness (3.5 cm) of the footpad. However, because of the 45° angle restriction on the foot joint, the maximum reach of *AquaRobot* is as shown in Figure 4-4. The CB, when the foot is at its maximum unconstrained reach of 178.2107 centimeters, is at 70.2107 centimeters off the terrain. Figure 4-4 includes an illustration of what would be the unconstrained vertical workspace if the maximum reach were used.

If the 45° foot joint angle is used again as the restricting dimension, it is possible for the minimum unconstrained reach of *AquaRobot* to become *negative*, in the sense that the foot can pass completely under the CB, as shown in Figure 4-5. This range of reach allows greater flexibility when designing a gait, but significantly increases the complexity of the gait algorithms because leg collision avoidance must be considered. Figure 4-5 includes a drawing of what would be the unconstrained vertical workspace if the minimum reach were used.

Three Matlab programs were written to verify the derivation of the kinematics transformations. Program ucwlink1.m, the results of which are shown in Figure 4-6, calculates and plots the unconstrained workspace of LINK1, the HIP to KNEE1 link. Figure 4-7 shows the results of program ucwlink2.m, which calculates and plots the unconstrained workspace of LINK2, the KNEE1 to KNEE2 link. Program ucwlink3.m, the results of which are shown in Figure 4-8, calculates and plots the unconstrained

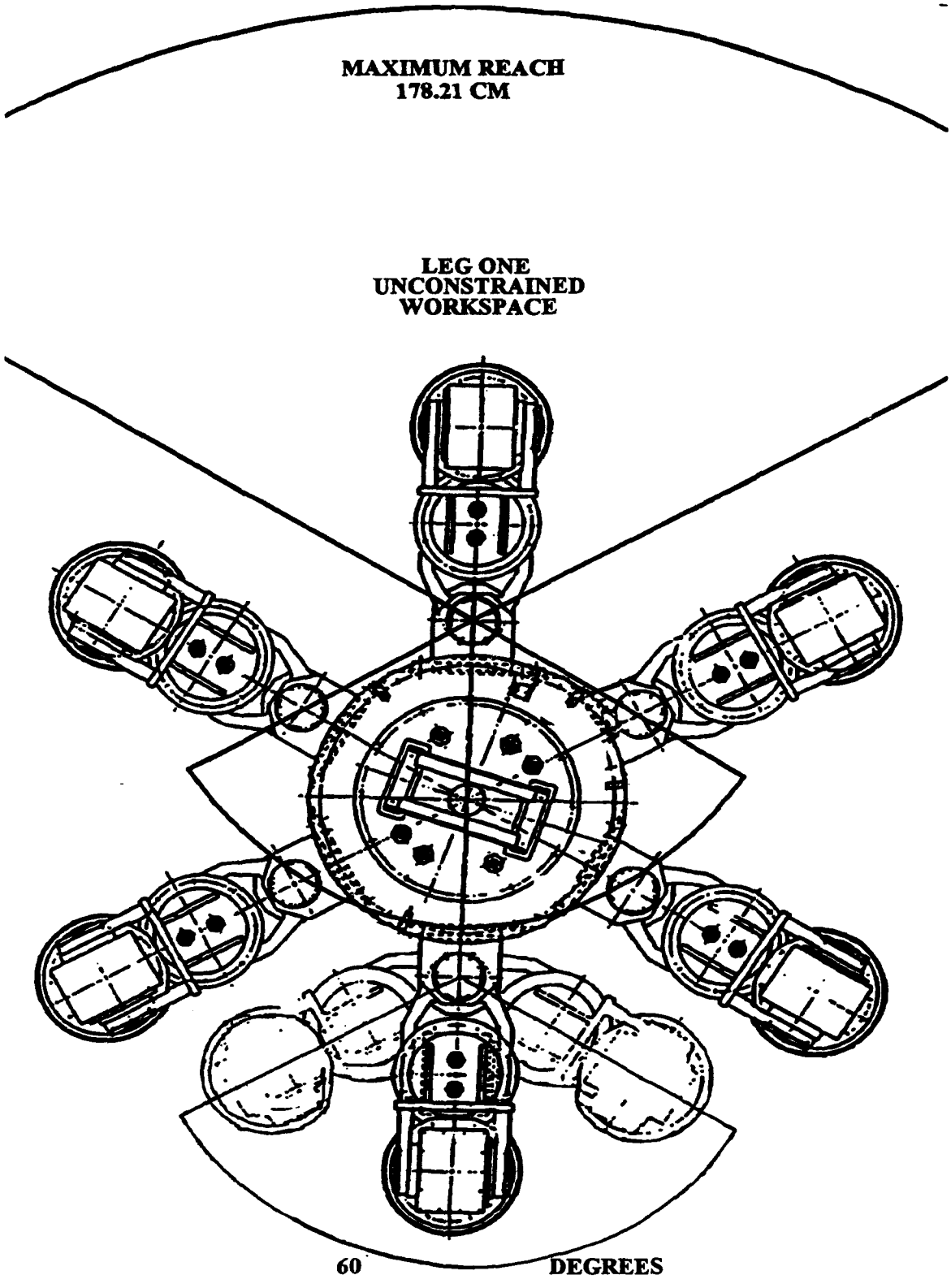


Figure 4-3. Unconstrained Horizontal Workspace.

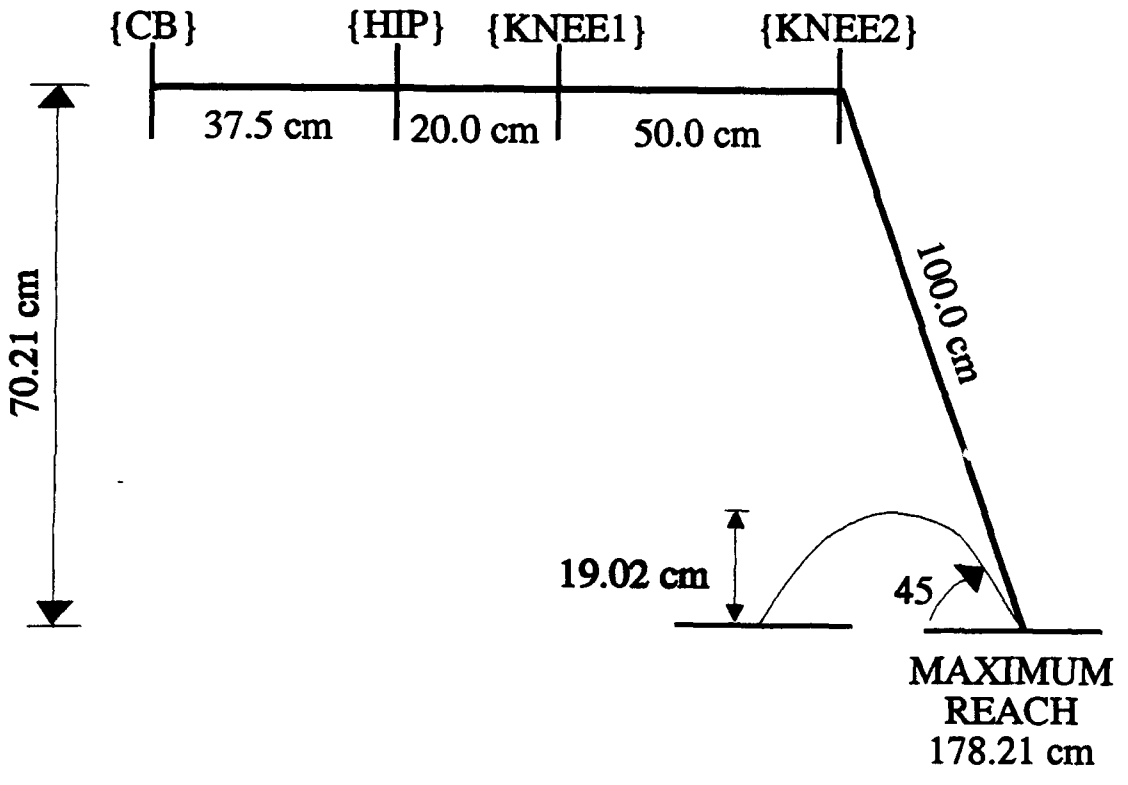


Figure 4-4. Maximum Reach and Associated Unconstrained Vertical Workspace.

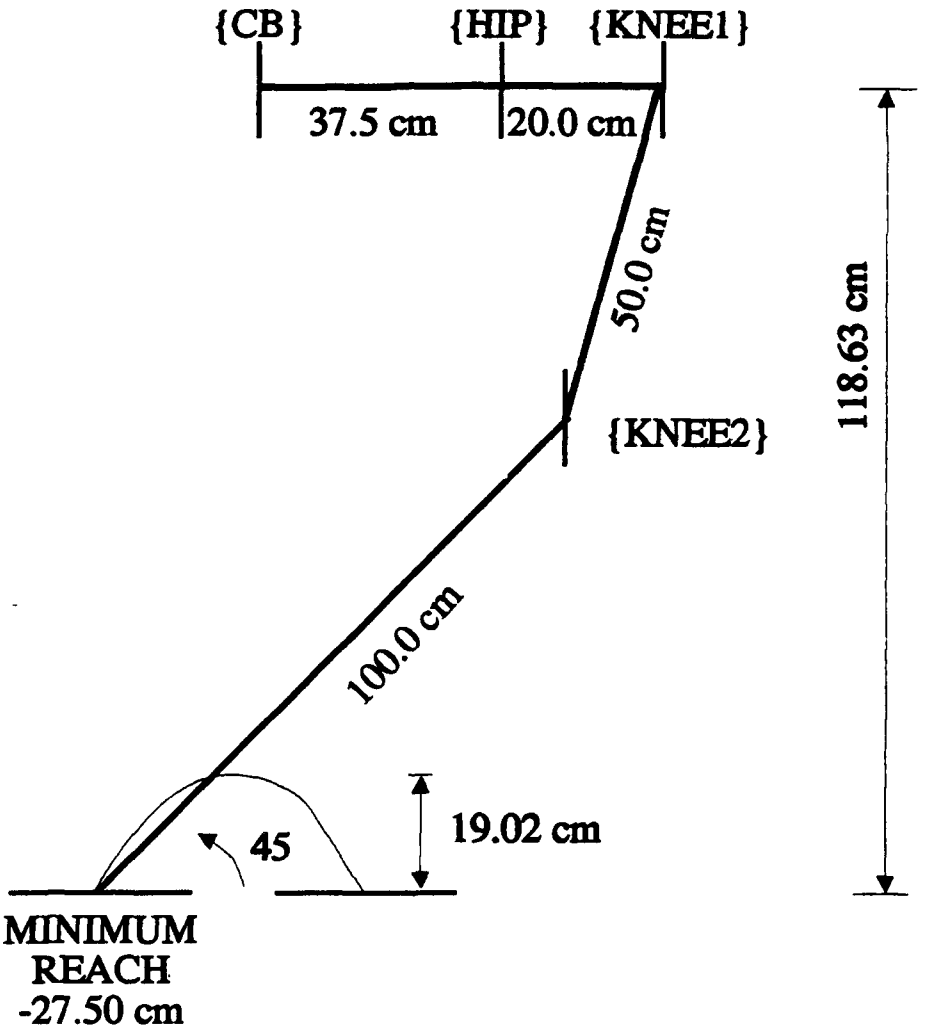


Figure 4-5. Minimum Reach and Associated Unconstrained Vertical Workspace.

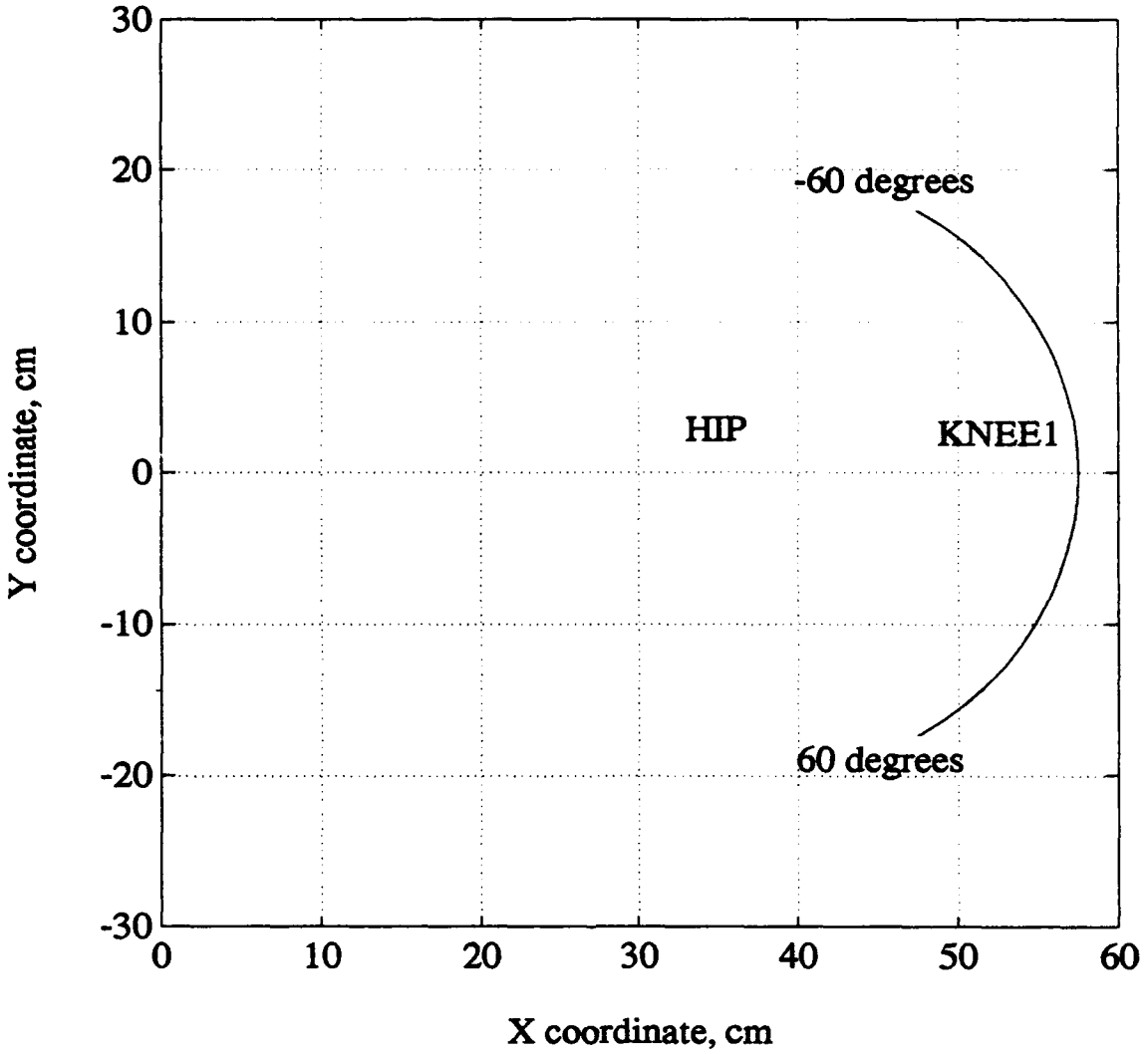


Figure 4-6. LINK1 Unconstrained Workspace.

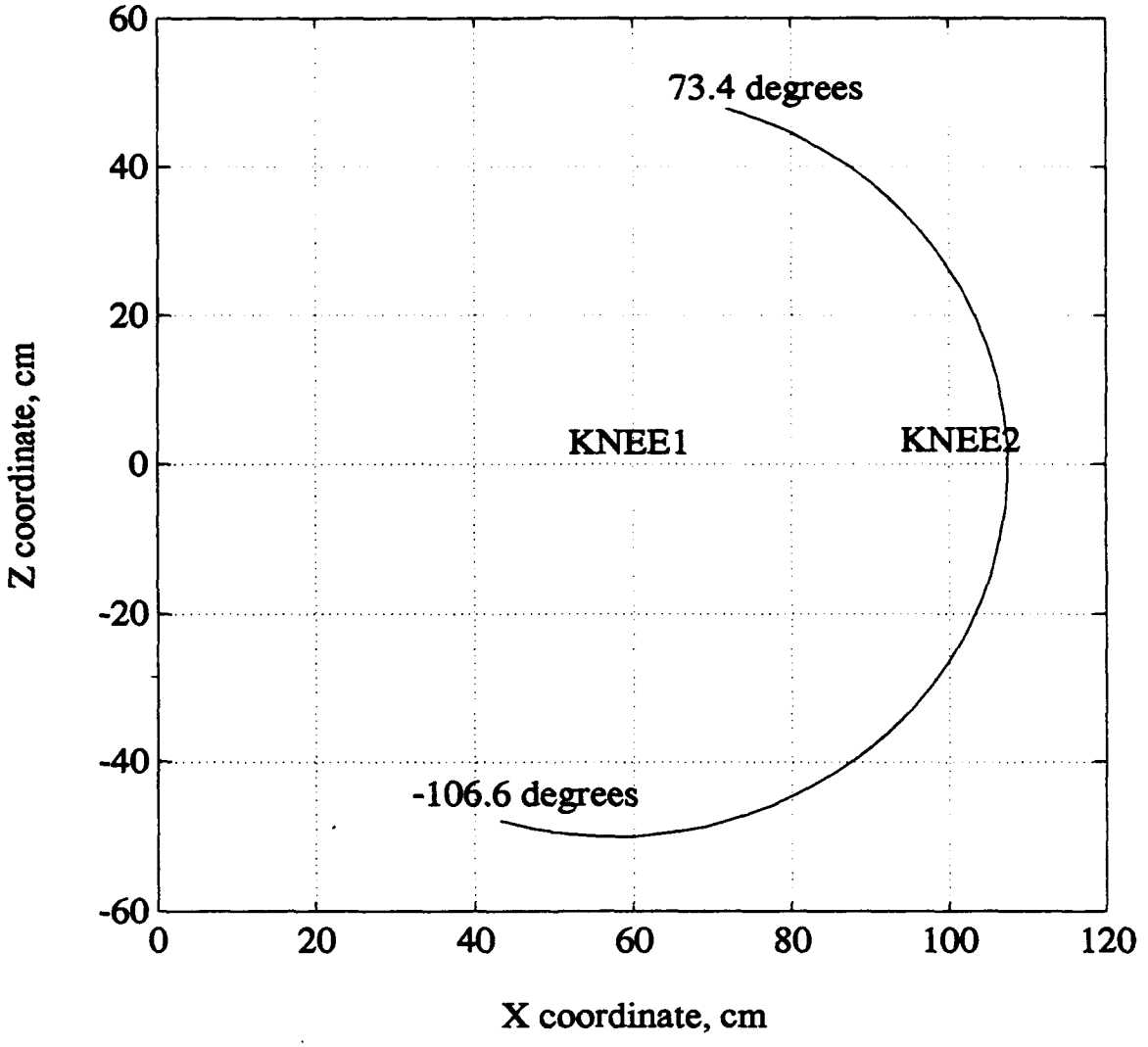


Figure 4-7. LINK2 Unconstrained Workspace.

workspace of LINK3, the KNEE2 to FOOT link. These three code modules are included in Appendix B.

2. Constrained Workspace

Now that the UWV is defined, we can define the CWV. The CWV used in this thesis results from six basic constraints:

- the CB height is fixed at the point where the maximum reach is achievable,
- the maximum and minimum reach meet the 45° foot joint angle requirement,
- the terrain is flat,
- the minimum reach is set equal to the radius the HIPs are from the CB,
- the legs are not allowed to collide or overlap, and
- the area encompassed by the footpads is included in case feet are added in future work.

Start with the maximum reach obtained when the CB is at 70.2107 centimeters. If the CB is then constrained to this value, the minimum reach becomes 36.79 centimeters. For simplification, we make the minimum reach equivalent to the radius the HIPs are from the CB: 37.5 centimeters. The resulting maximum and minimum constrained reach is shown in Figure 4-9 and represents the constrained vertical workspace. The height that the leg can be raised is obviously restricted when the CB height is fixed. However, this factor is not as important, when using flat terrain, as reducing the number of variables when first implementing a control program. Changing the CB height to allow traversal of unstructured terrain is easily accomplished if required in future research.

Figure 4-10 shows the constrained horizontal workspace. The constrained horizontal workspace is developed using LEG1, LEG2, and LEG6 as an example. Each LEG is drawn with its centerline beginning at the CB and bisecting the ± 60° joint limit of its HIP. The boundary for the excluded areas is selected as lines that separate the

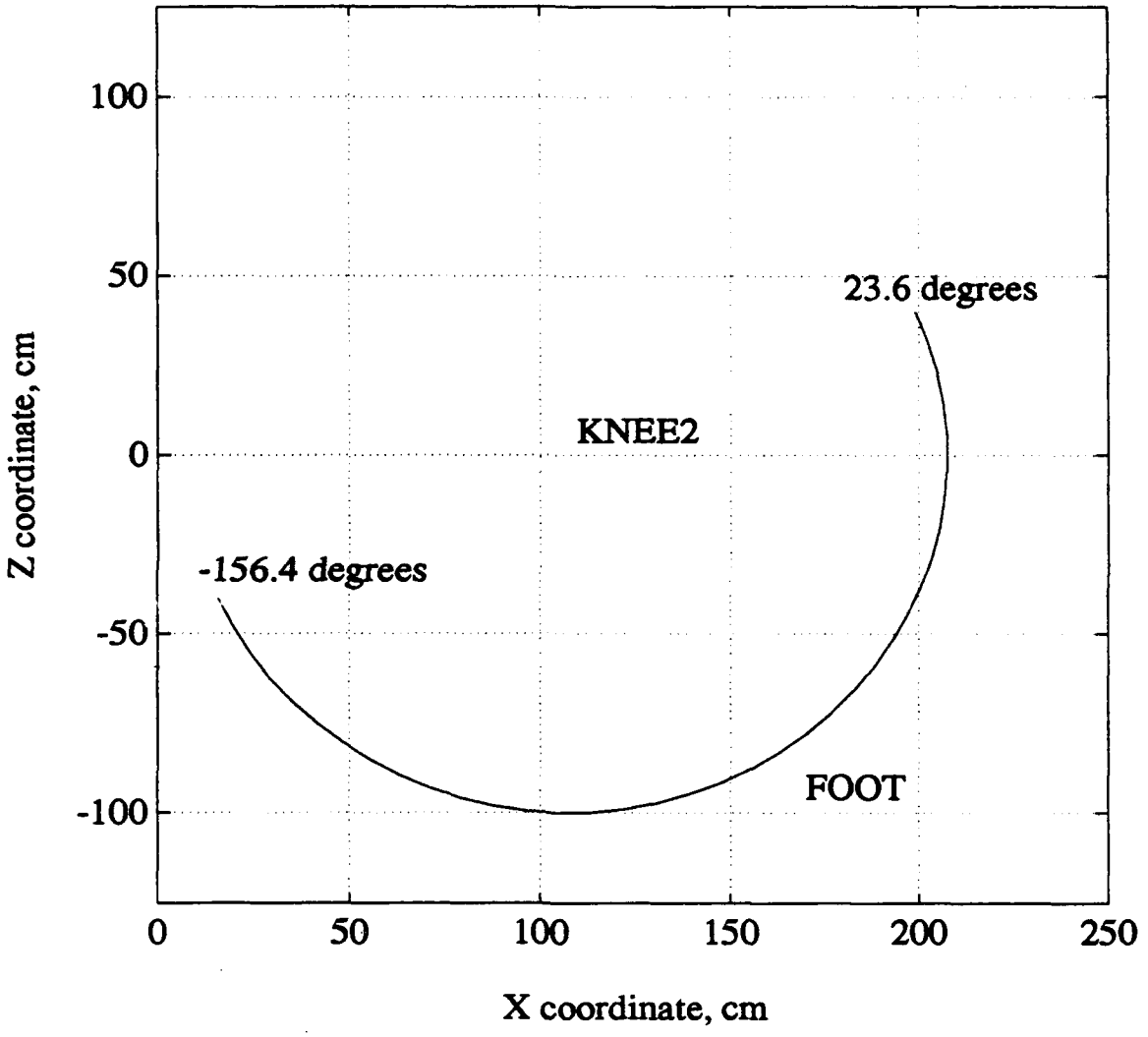


Figure 4-8. LINK3 Unconstrained Workspace.

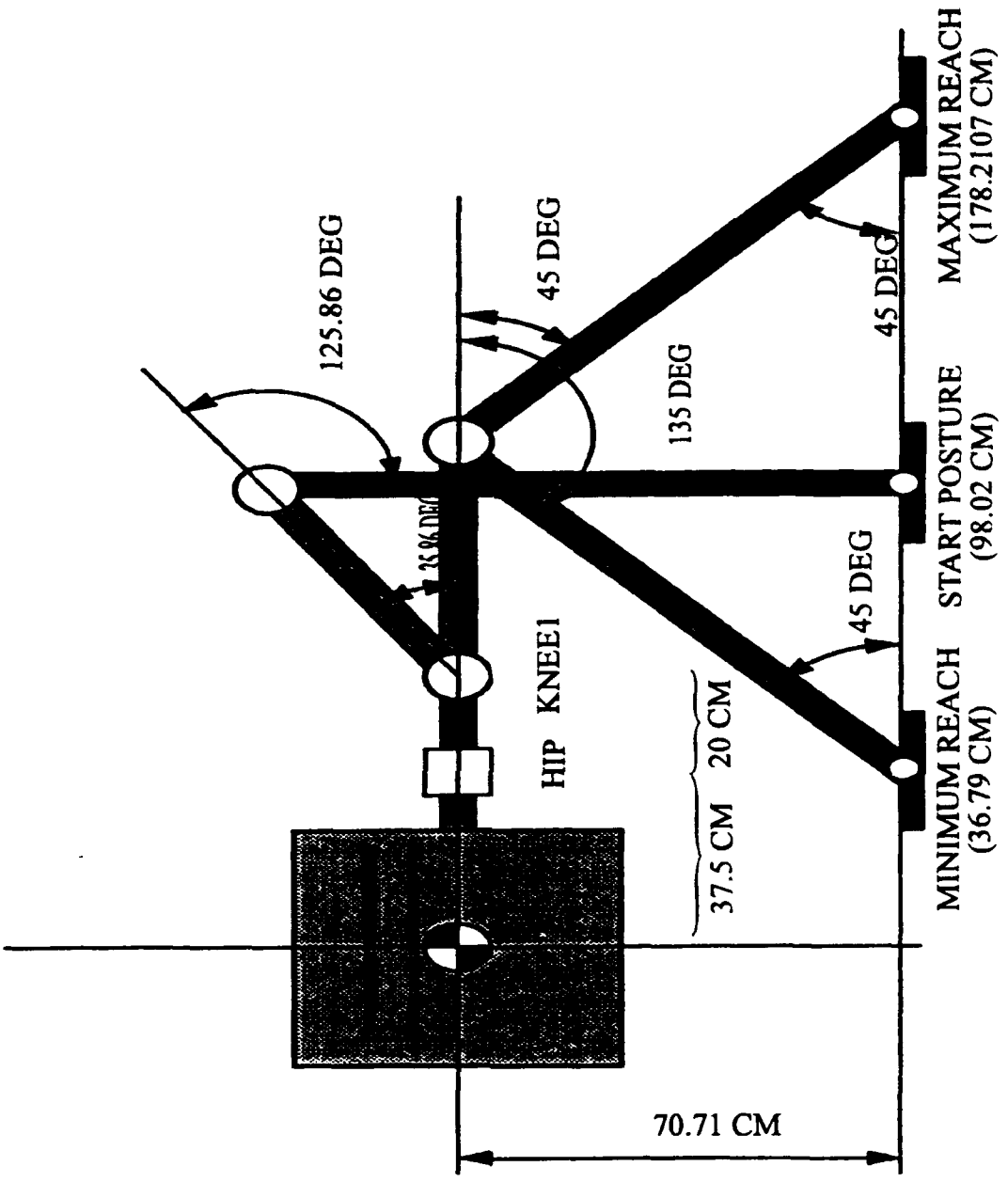


Figure 4-9. Maximum and Minimum Constrained Reach.

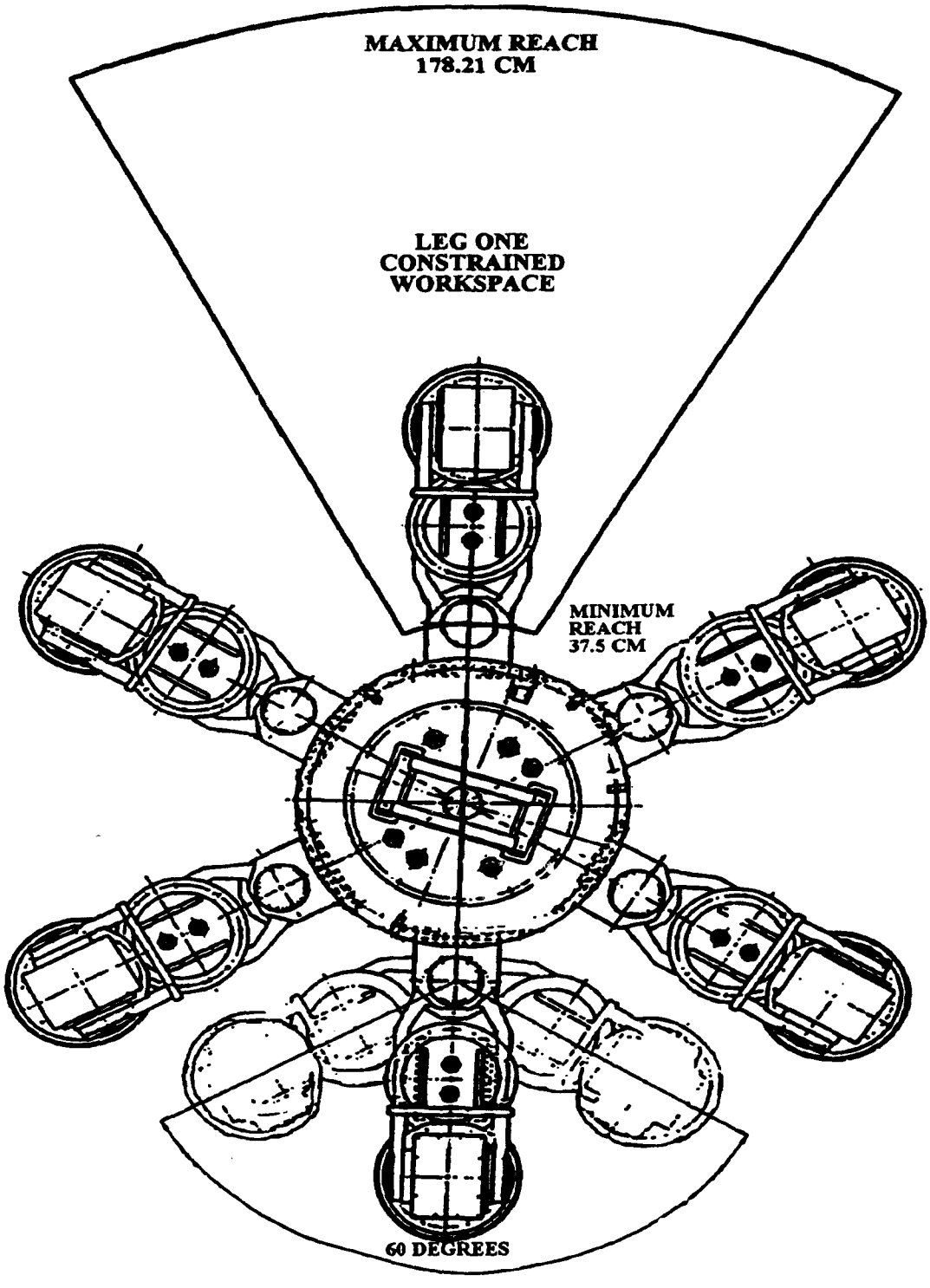


Figure 4-10. Constrained Horizontal Workspace.

reachable areas for the three legs exactly. The actual exclusion areas are then chosen based on the edge of the footpad touching these separation lines. This method allows the addition of the footpads in the future without changing the workspace algorithms. The workspace boundaries then form the sides of a section of an annulus circumscribed about the CB. The inner boundary of the annulus is the radius the HIPs make about the CB. The outer boundary is the radius of the maximum reach of a LEG about the CB.

3. Strategy for Constrained Workspace Use

The primary purpose of the constrained workspace is to define the limits for determining the possible length of the stride. If an arbitrary direction is chosen for a representative foot to travel, that foot will intersect the edge of the constrained workspace at some point. Determining the point of intersection requires algorithms to find the intersection of a directed line and a line segment and the intersection of a directed line and a section of a circle. Further, if an intersection is found, it must be the intersection at the correct orientation of the directed line. Development and implementation of methods to generate the required results are abstracted from Kanayama [KAN92].

First, we develop the method to determine the intersection of two directed lines. In our case, one directed line is derived from the current foot position and desired direction of travel. The second directed line is defined as a side of the constrained workspace for a given leg, LEG1. Referring to Figure 4-11, we choose an arbitrary direction of θ_1 for the foot to travel within the LEG1 workspace. The current position of the foot is presumed to have coordinates $p_F = (x_B, y_B)$, with no z_B component because only the horizontal workspace is of interest at this point. The directed line q_1 is then formed using the expression

$$q_1 = \{(x_1, y_1), \theta_1\}, \tag{4.34}$$

where $x_1 = x_B$ and $y_1 = y_B$.

We know the endpoints of the line segments that form the sides of the constrained workspace for LEG1. Using the known endpoints and the two-argument arctangent (Atan2) function, it is an easy task to find the angle the line segments make with the body-fixed coordinate system. If the angle for one side is taken as θ_2, the second directed line q_2 becomes

$$q_2 = \{(x_2, y_2), \theta_2\}. \tag{4.35}$$

If we let an intersection be (x, y), the distance between the intersection point and either of the two directed lines is zero.

$$(y - y_1)\cos\theta_1 - (x - x_1)\sin\theta_1 = 0 \tag{4.36}$$

$$(y - y_2)\cos\theta_2 - (x - x_2)\sin\theta_2 = 0 \tag{4.37}$$

Combining equations 4.36 and 4.37 into matrix form:

$$\begin{bmatrix} \sin\theta_1 & -\cos\theta_1 \\ \sin\theta_2 & -\cos\theta_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_1 \sin\theta_1 - y_1 \cos\theta_1 \\ x_2 \sin\theta_2 - y_2 \cos\theta_2 \end{bmatrix}. \tag{4.38}$$

Then, solving equations 4.36 and 4.37 simultaneously yields the intersection point

$$\begin{aligned} x_{intercept} &= \frac{1}{\sin(\theta_2 - \theta_1)} \begin{vmatrix} (x_1 \sin\theta_1 - y_1 \cos\theta_1) & -\cos\theta_1 \\ (x_2 \sin\theta_2 - y_2 \cos\theta_2) & -\cos\theta_2 \end{vmatrix} \\ &= \frac{-\cos\theta_2 (x_1 \sin\theta_1 - y_1 \cos\theta) + \cos\theta_1 (x_2 \sin\theta_2 - y_2 \cos\theta_2)}{\sin(\theta_2 - \theta_1)} \end{aligned} \tag{4.39}$$

and

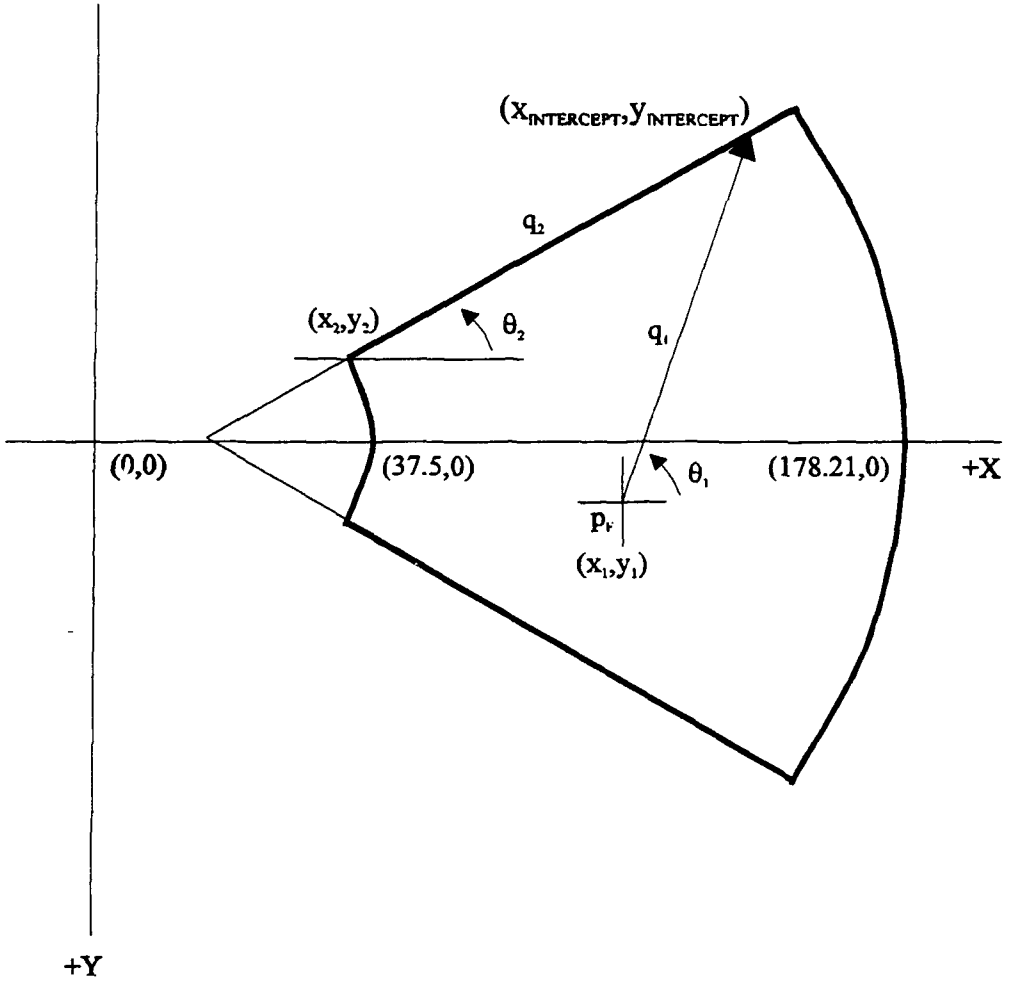


Figure 4-11. Intersection of Directed Line and Line Segment.

$$y_{intercept} = \frac{1}{\sin(\theta_2 - \theta_1)} \begin{vmatrix} \sin\theta_1 & (x_1 \sin\theta_1 - y_1 \cos\theta_1) \\ \sin\theta_2 & (x_2 \sin\theta_2 - y_2 \cos\theta_2) \end{vmatrix}$$

$$= \frac{\sin\theta_1(x_2 \sin\theta_2 - y_2 \cos\theta_2) - \sin\theta_2(x_1 \sin\theta_1 - y_1 \cos\theta_1)}{\sin(\theta_2 - \theta_1)}. \tag{4.40}$$

The intersection point is then tested to ensure it lies within the desired segment of the directed line forming the boundary of the constrained workspace. Kanayama proposes that for any three arbitrary distinct points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, and $p_3 = (x_3, y_3)$ on a directed line, p_1 is *upstream* of p_2 and p_2 is *upstream* of p_3 if and only if

$$x_1 \cos\alpha + y_1 \sin\alpha < x_2 \cos\alpha + y_2 \sin\alpha < x_3 \cos\alpha + y_3 \sin\alpha, \tag{4.41}$$

where α is the orientation of the directed line that includes the three points [KAN92]. The endpoints for the line segment in question are substituted for points p_1 and p_3 and the intersection point is substituted for point p_2 in equation 4.41. Alpha (α) then becomes the angle associated with the directed line that makes up the side of the constrained workspace. From these values, we determine whether the intersection point is within the endpoints of the line segment of interest. The Matlab program segint.m, found in Appendix B, performs the segment intersection calculations described above.

Now we turn our attention to developing the method to determine the intersection of a directed line and a circular section. Again, a directed line is derived from the current foot position and desired direction of travel. The circular section is defined as either the inner or outer radius of the constrained workspace for a given leg, LEG1. In this case, we must test for an intersection and then determine whether the intersection is within the section of interest. Referring to Figure 4-12, we choose an arbitrary direction of α for the foot to travel within the workspace. The coordinates and orientation of the foot are

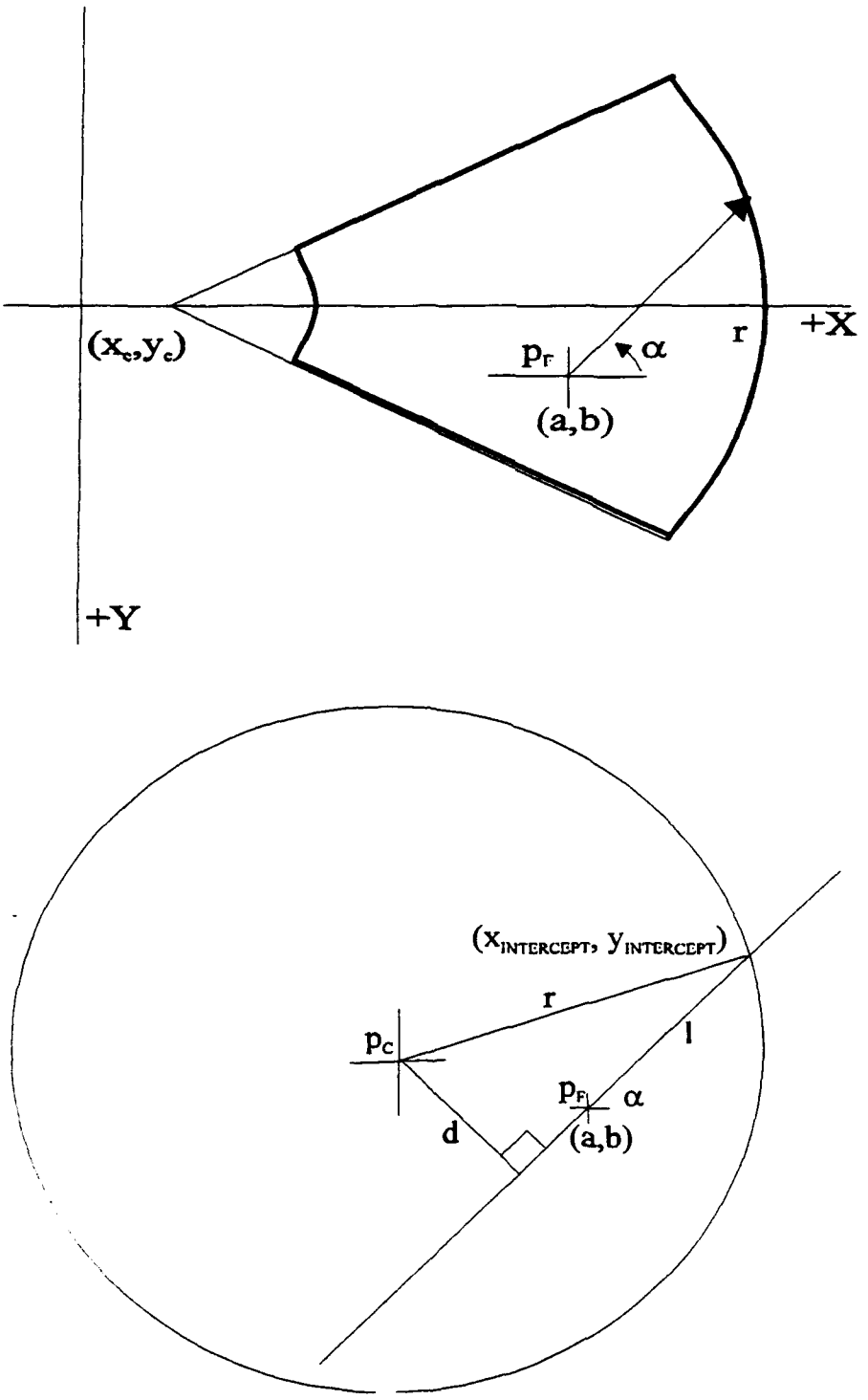


Figure 4-12. Intersection of Directed Line and Arc.

then known: $p_F = \{(a,b),\alpha\}$. The radius and center of the circle of interest are then defined, respectively, as r and $p_C = (x_C, y_C)$. From these values, we can determine the perpendicular distance d from p_C to the directed line previously described by p_F. using the relationship

$$d = (y_C - b)\cos\alpha - (x_C - a)\sin\alpha. \tag{4.42}$$

The radius r and the perpendicular distance d then make up two sides of a right triangle with side l unknown. Side l is found using the Pythagorean Theorem:

$$l = \sqrt{r^2 - d^2}. \tag{4.43}$$

Next, the point $p_1 = (x_1, y_1)$, where d and l intersect is found using

$$x_1 = x_C + d\cos\left(\frac{\pi}{2} - \alpha\right) = x_C + d\sin\alpha \tag{4.44}$$

and

$$y_1 = y_C - d\sin\left(\frac{\pi}{2} - \alpha\right) = y_C - d\cos\alpha. \tag{4.45}$$

The results of equations 4.43, 4.44, and 4.45 are then used to find the intersection point of the directed line and the circular section using

$$x_{intercept} = x_1 + l\cos\alpha \tag{4.46}$$

$$y_{intercept} = y_1 + l\sin\alpha \tag{4.47}$$

if the foot is located inside the diameter of the circle and

$$x_{intercept} = x_1 - l\cos\alpha \tag{4.48}$$

$$y_{intercept} = y_1 - l\cos\alpha \tag{4.49}$$

if the foot is located outside the diameter of the circle.

Because the ends of the circular sections of the constrained workspace are known, the intersection point is then tested to ensure it lies within the section of interest using the method described in equation 4.41. The Matlab program arcint2.m, found in Appendix B, performs the arc intersection calculations just described.

Once the intersections of the various feet and their respective workspaces are known, the *maximum stride* is determined. The maximum stride is defined as the *minimum* of the maximum strides of the individual feet in a given tripod. Using the maximum stride allows all three feet in the tripod to move the same distance in the same amount of time. Figure 4-13 illustrates finding the maximum stride possible, given the desired direction of travel is –45° and the tripod selected is TRIPOD0 (LEGs 1, 3, 5). The Matlab program maxd25.m, found in Appendix B, determins the maximum stride.

E. TERRAIN AND POSTURES

1. Terrain Considerations

A walking machine is expected to operate in structured or unstructured terrain. The terms structured and unstructured are too diverse to use when describing the possible classification and description of terrain. Hirose suggests using five different types of terrain classification, as shown in Figure 4-14. First, the O type terrain is a surface which supports standard rhythmical walking. The flat terrain model used in this study is of type O. Second, the H type terrain includes deep holes in which the legs of the walking machine can not reach bottom. A walking machine can not choose the holes as possible footholds, but can step over the holes. Next, the P type terrain is described as having poles or rocks higher than the walking vehicle itself. For P type terrain, the walking machine must avoid trying to cross over the poles and rocks. HP terrain is a combination of the H and P terrain types. Finally, the G type is considered the usual rough terrain. [HIR86a]

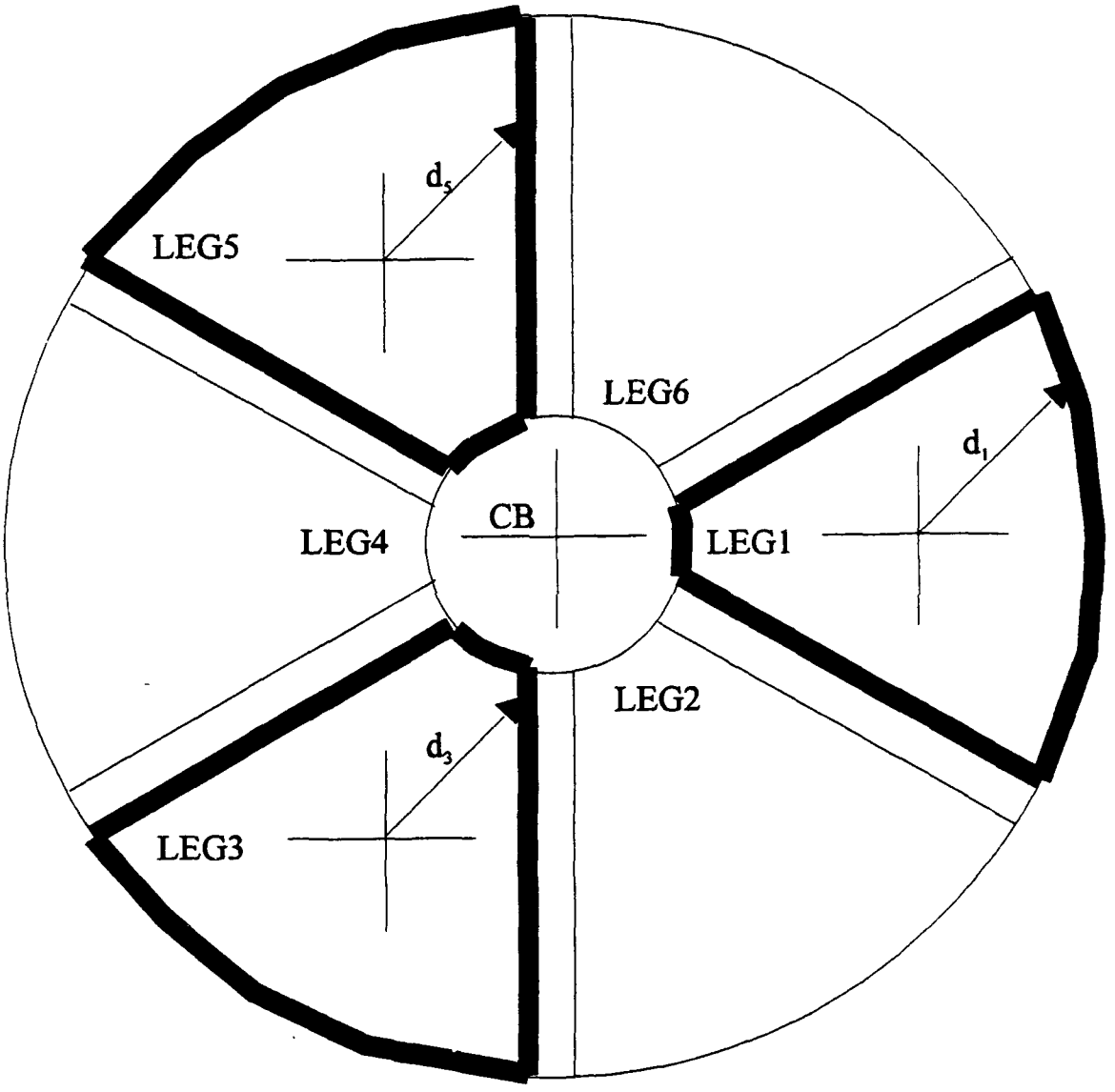


Figure 4-13. Determination of Maximum Stride.

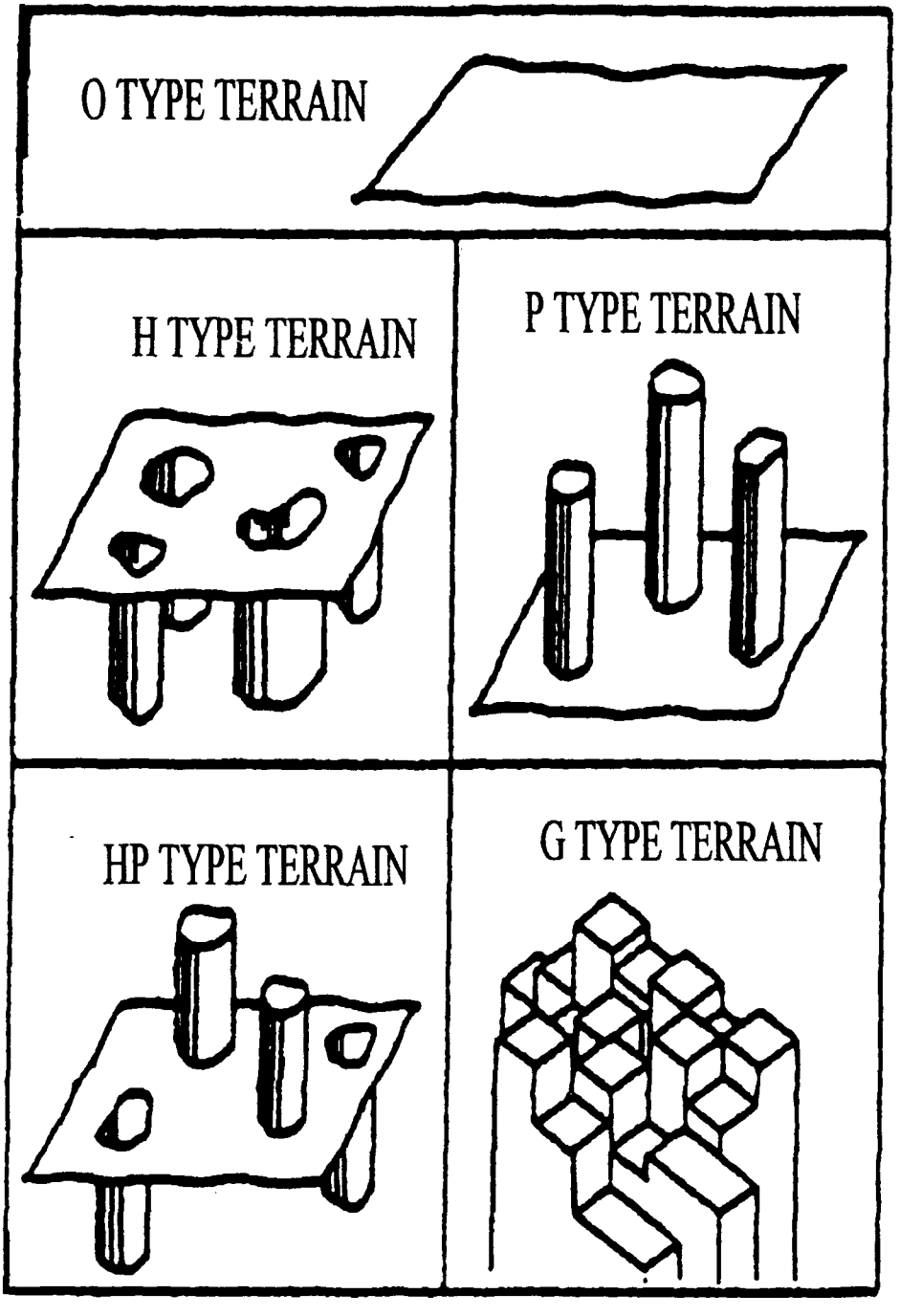


Figure 4-14. Terrain Classification (After [HIR86a]).

Basically, the terrain models for *AquaRobot* fall within three general categories: 1) essentially flat, O type terrain for this study, 2) G type terrain with random irregularities of ± 35 centimeters to simulate walking on an unfinished rubble mound, and 3) O type terrain with random undulations of ± 5 centimeters to simulate completed rubble mound walking.

2. Operational Postures

AquaRobot was designed to walk using body postures that optimize the particular task or function it is performing. These *modes* of operation vary according to stability, speed, and terrain requirements. There are essentially five modes of operation generally used:

- storage mode,
- initialization mode,
- walking mode,
- height and width change mode, and
- slim mode.

AquaRobot requires support when no electricity is supplied to its joints because there are no brakes. Hence, *AquaRobot* rides on a mechanical support platform with its legs folded so the minimum volume is used. When *AquaRobot* has its legs folded on the support platform, it is in the *storage mode*. The storage mode posture is termed *RESET* in this study. When *AquaRobot* is in the storage mode, the incremental encoders of the DC motors are set to an initial position. The FOOT positions for the RESET posture, plotted from the Matlab program reset.m, are as shown in Figure 4-15. [IWA87]

During the *initialization mode* of operation, *AquaRobot* spreads its legs, steps off the support platform, and assumes an arbitrary *START* posture [IWA87]. We define the START posture in this study based on the maximum reach and the constrained

workspace. Therefore, the START posture allows *AquaRobot* to achieve the maximum possible stride while maintaining an adequate clearance over the flat terrain. The FOOT positions for the START posture, plotted from the Matlab program start.m, are as shown in Figure 4-16.

The *walking mode* is subdivided into two modes: 1) flat terrain walking mode, and 2) irregular terrain walking mode. Only the flat terrain walking mode is used in this study. The *height and width change mode* allows changing the stance of *AquaRobot* based on the type of terrain and the water speed. The *tall* posture is a narrower stance used for rougher terrain. This means the control algorithm can raise the feet higher to step over higher obstacles. The *short* posture is a wider stance used when the current is fast. The short posture lowers the vehicle's center of gravity and provides more stability in these situations. The tall and short postures are attained simply through adjusting the vehicle's center-of-body height over the terrain. [IWA87]

The *slim mode*, which is not used in this study, was developed to allow *AquaRobot* to pass between closely spaced objects. In the slim mode, the body width is decreased to one-half of its original size by rotating some legs so bilateral symmetry is achieved. *AquaRobot* takes on the posture of a crab and uses a slim version of the alternating tripod gait to maneuver. [IWA87]

F. SUMMARY

The central topic of this chapter was developing and implementing kinematics and inverse kinematics for *AquaRobot*. Next, coordinate transformations required to move from body-to-world and from world-to-body coordinate systems were derived. Then, an acceptable constrained workspace was designed. Finally, *AquaRobot's* postures, as applied to various terrain types, were discussed. In the following chapter, static stability issues are addressed.

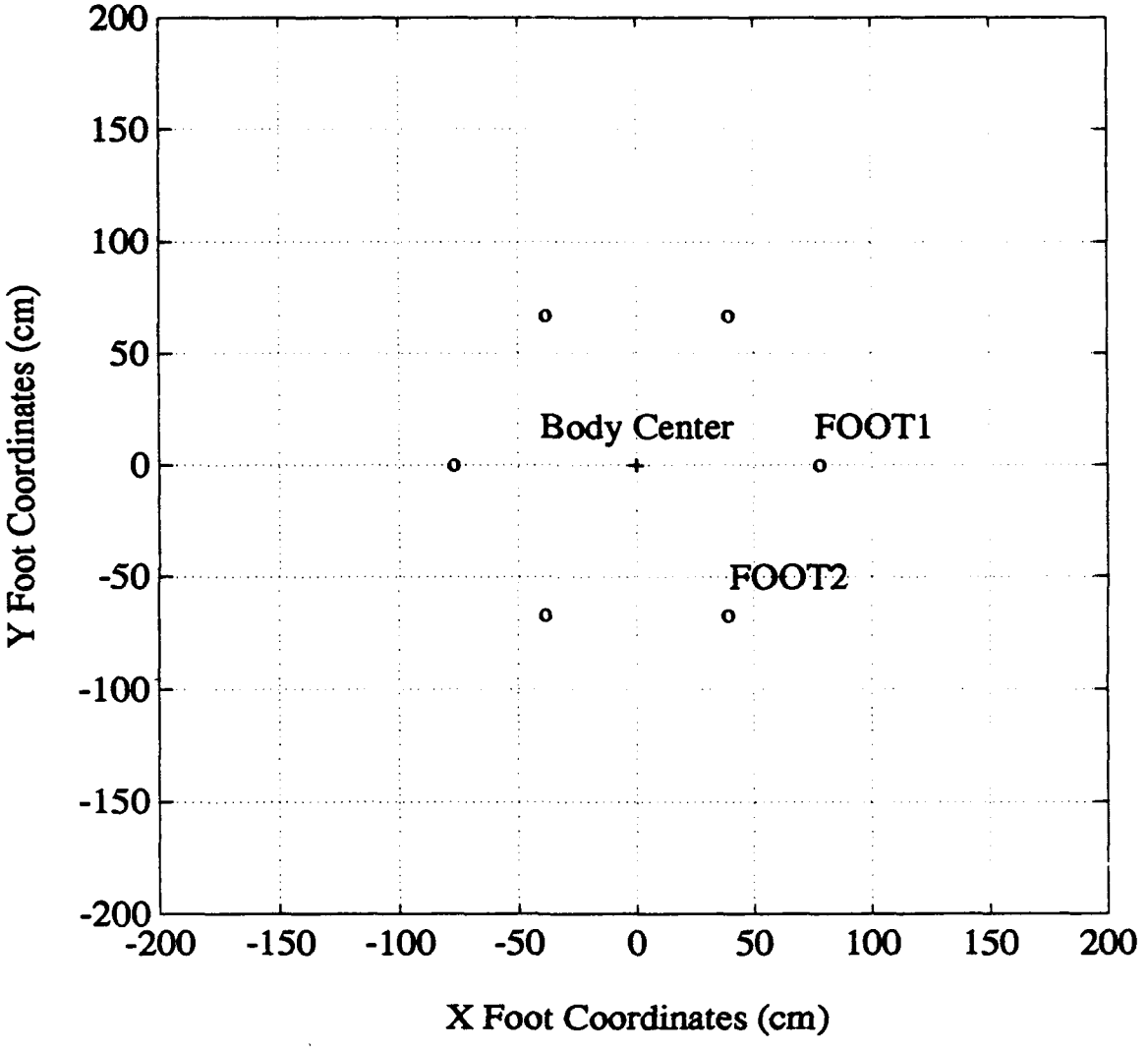


Figure 4-15. ***AquaRobot*** **RESET Posture.**

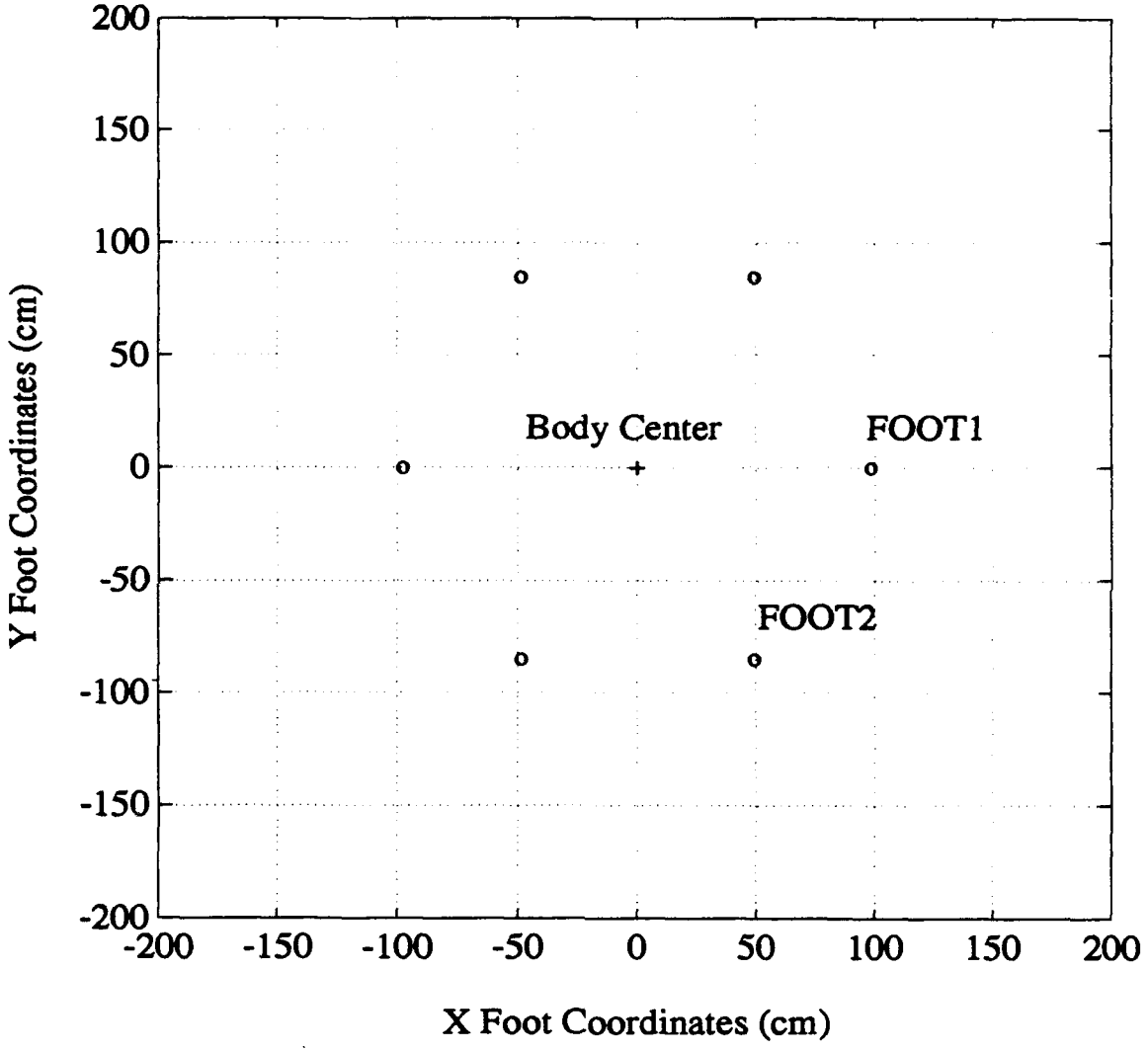


Figure 4-16. *AquaRobot* START Posture.

V. STABILITY

A. INTRODUCTION

One of the main considerations in walking machines is how to achieve stability, that is, to keep from falling over [TOD85]. In this study, the static stability concept is employed to provide a simple basis for the control of vehicle locomotion. The concept of stability margin is known as the yardstick for measuring static stability. The value is given in the relation between the center of gravity and the polygon of supporting legs, and is also governed by the height of the center of gravity and compliance of the legs [HIR86a]. For simplicity, however, this study uses the longitudinal stability margin S_l as the primary approximation of stability.

In general, it is possible for the center of gravity to go outside the support pattern at some instants during locomotion, even if the support joints are constrained to be inside the working volumes of the legs [LEE88b]. However, there is some flexibility in changing the position and dimensions of the CWV inside the original working volume. This flexibility is used to increase vehicle stability [LEE88a]. In our case, the footpad of *AquaRobot* is actually 25 centimeters in diameter, yet we are using a point foot representation. This gives us an additional 12.5 centimeters of stability safety margin. As such, mild accelerations and decelerations can be accomodated without changing the existing algorithms.

From the previous chapter, we know how to find the maximum stride possible given the constraints of the workspace and the mechanical joint limitations. It is possible that the maximum stride would allow the CG to move beyond the support pattern. Using the

maximum stride alone, then, would result in unstable walking. Therefore, the stride and stability calculations are compared for each increment of motion and the lesser of the two is chosen as the distance the foot will travel. Although this method greatly reduces the amount of motion flexibility, it ensures stability and a reasonably fast gait.

1. Center-of-Body Versus Center-of-Gravity

The center of *AquaRobot's* body is located at the intersection of the plane formed by its HIPs and the vertical center of the torso. The CB remains constant during any body or leg motion. *AquaRobot's* center-of-gravity, however, changes with each incremental motion of its legs or body. Therefore, at any instant in time, the CG is not necessarily located at the CB.

The PHRI had not completed CG calculations at the time of this thesis writing. Because the CG is only needed for stability purposes, the CG is set equal to the CB in this study. Once the CG calculations are complete, the CG is easily substituted for the CB in stability algorithms.

2. Static Versus Dynamic Stability

Static and dynamic stability distinguishes the two types of walking machines that have been the subjects of recent research. Statically stable systems resist falling down by trying never to get in a situation where falling down is possible. Dynamically stable systems balance to keep from falling down. Statically stable machines have at least four legs and sometimes as many as eight, though more typically six. Dynamically stable machines have from one to four legs. The main difference is the use of balance in the control of body attitude in dynamically stable machines. [DON87]

Of static and dynamic walking, static locomotion is more basic and less complicated than dynamic locomotion in terms of control algorithms and motion analysis [HIR86a]. It turns out that insects and other many-legged creatures walk with statically

stable gaits [DON87]. *AquaRobot* is an example of an "insect type" machine. Higher animals, like horses, cats, and people, all walk using dynamically stable gaits [DON87]. The *AquaRobot* simulation presented in this thesis uses a statically stable gait algorithm.

B. STABILITY MODEL

The stability model used in this study is similar to that proposed by McGhee [MCG86]. The longitudinal stability margin S_l is defined as the shortest distance from the body's center of gravity to the boundary of the support pattern, measured in the direction of travel. From the definition of longitudinal stability margin, a gait is deemed statically stable if $S_l \geq 0$. Otherwise, it is deemed statically unstable. Vehicle stability, stability intercepts, and stability margin are determined using the known foot positions of the supporting tripod and the known body center. All stability calculations use the body-fixed coordinate system and substitute the CB for the CG.

Prior to planning any new leg motion, *AquaRobot's* stability is evaluated. The evaluation consists simply of determining whether the vertical projection of the CB lies within an arbitrary support pattern formed from the vertical projection of the three feet in a selected tripod. Referring to Figure 5-1, we see the relationship of the supporting polygon, in this case, a triangle, to the CB at a particular instant in time for TRIPOD0 (LEGs 1, 3, 5) and TRIPOD1 (LEGs 2, 4, 6). The CB is represented by the point $p_1 = (x_1, y_1)$ and the feet are represented by the points $p_2 = (x_2, y_2)$, $p_3 = (x_3, y_3)$, and $p_4 = (x_4, y_4)$. Point p_2 is always assigned to either FOOT1 or FOOT2, depending on whether TRIPOD0 of TRIPOD1, respectively, is selected. Point p_3 is always assigned to either FOOT3 or FOOT4 and point p_4 is always assigned to either FOOT5 or FOOT6. The CB and feet, then, divide the support polygon into three smaller triangles.

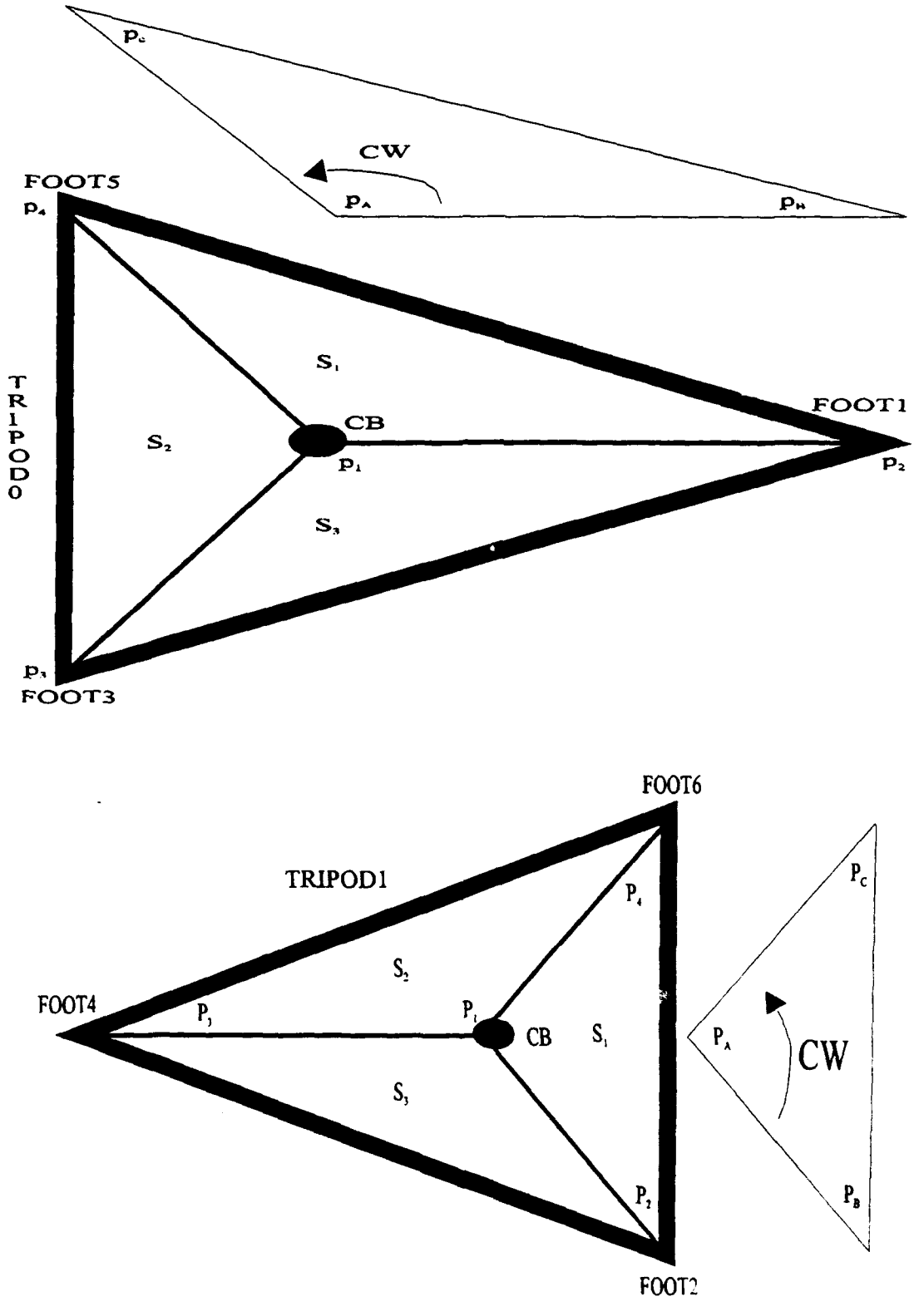


Figure 5-1. Stable TRIPODs.

Kanayama defines three *modes* of a triple of points: counterclockwise (CCW), clockwise (CW), and colinear [KAN92]. A counterclockwise mode results when the *order* of the points is as shown in Figure 5-2(a). A clockwise mode results when the order of the points is as shown in Figure 5-2(b). With these definitions in mind, we can describe the area, and thereby, the stability, of a triangle as a triple of distinct points such that

$$S(p_1, p_2, p_3) \equiv \frac{1}{2}\begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix}. \tag{5.1}$$

Expanding equation 5.1 yields:

$$S(p_1, p_2, p_3) = \frac{1}{2}\left[(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)\right]. \tag{5.2}$$

The orientation of points is always chosen such that:

- if the CB is inside the supporting polygon, the order is CCW and S is positive, or
- if the CB is outside the supporting polygon, the order is CW and S is negative, or
- if the CB is on the supporting polygon, the order is colinear and S is zero.

If S is ≥ zero, the tripod is considered stable. Figure 5-1 illustrates a statically stable support pattern. If S < zero, the tripod is considered unstable. Figure 5-3 illustrates a statically unstable support pattern. The safety margin of 12.5 centimeters discussed earlier is not implicitly used in the evaluation of stability. The Matlab function stable.m, found in Appendix B, performs the stability test described above. [KAN92]

Once the stability is ascertained, the stability intercepts and longitudinal stability margin S_l are found. The stability intercepts are determined using Kanayama's method for finding the intersection of two directed lines, discussed in Chapter IV. The stability margin is simply the distance between the CB coordinates and the stability intercepts in the

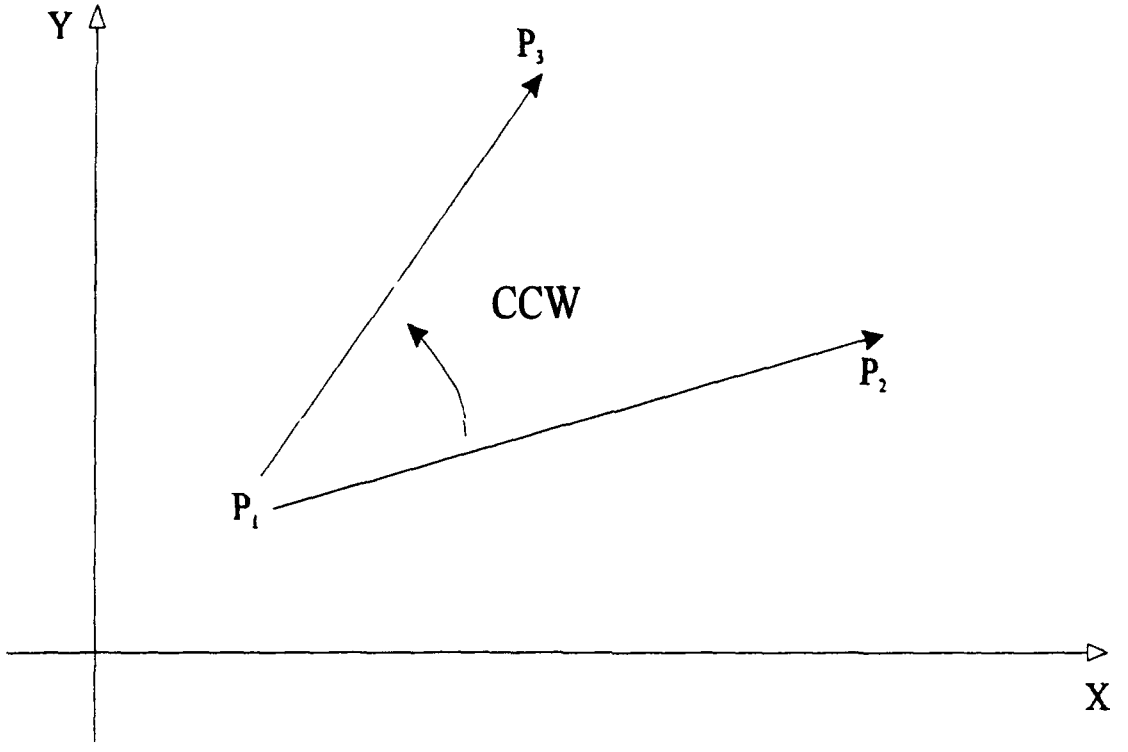


Figure 5-2(a). Counterclockwise Mode of a Triple of Points (After [KAN92]).

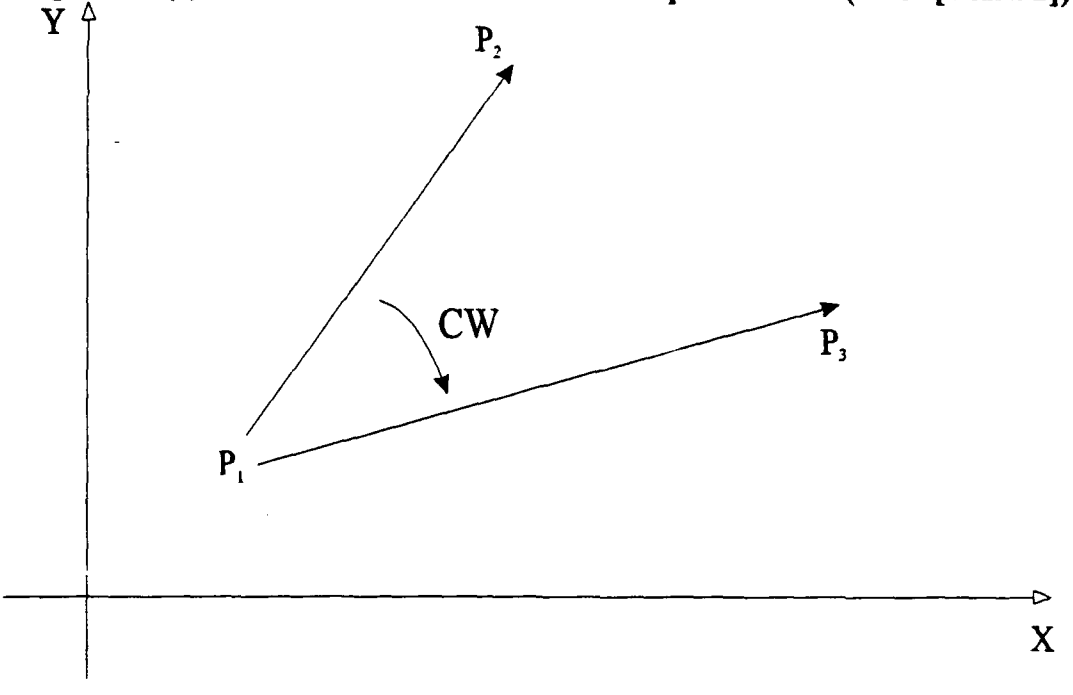


Figure 5-2(b). Clockwise Mode of a Triple of Points (After [KAN92]).

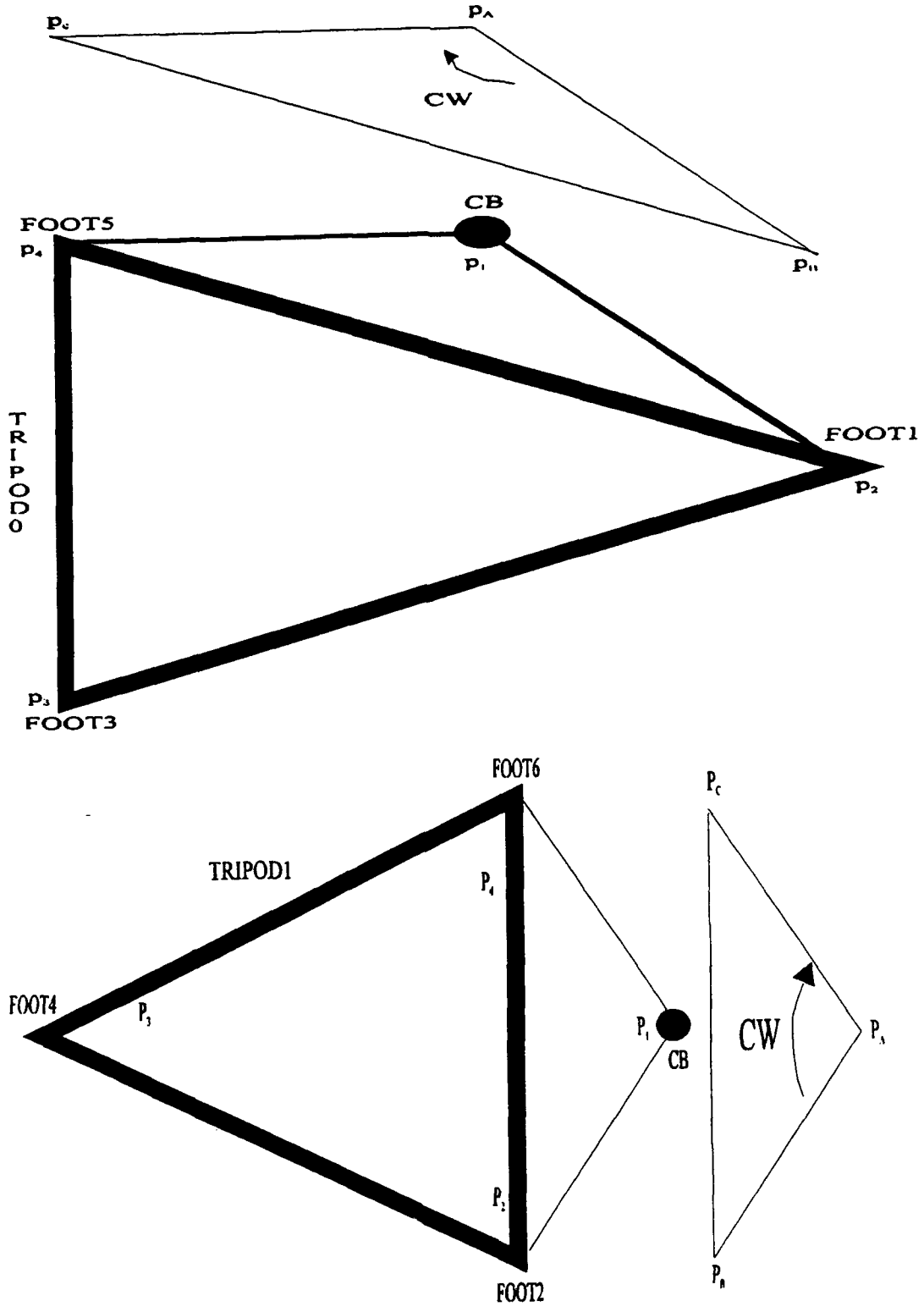


Figure 5-3. Unstable TRIPODs.

direction of travel:

$$S_l = \sqrt{\left(y_{intercept} - CB_y\right)^2 + \left(x_{intercept} - CB_x\right)^2} \quad (5.3)$$

The Matlab function stabint.m, found in Appendix B, finds the stability intercepts of a given tripod and determines the stability margin.

C. SUMMARY

This chapter described the concept of static stability as it pertains to *AquaRobot*. For simplification purposes, the body center was adopted as the vehicle's center of gravity. A method for determining static stability, including the longitudinal stability margin, was derived. The following chapter describes *AquaRobot's* current rectangular leg motion model and proposes two alternative smooth leg motion models: elliptical and cycloidal.

VI. SMOOTH FOOT MOTION

A. INTRODUCTION

Rough, jagged motion tends to cause abnormal wear of mechanical mechanisms, vibrations resulting from excited mechanical resonances, and inefficiencies of motor operation. Therefore, *smooth* motion is a desirable characteristic. A smooth motion function is one which is continuous and which has a continuous first derivative and, possibly, a continuous second derivative. Smooth motion for *AquaRobot* implies smooth foot and smooth body motion. This chapter is concerned with developing smooth foot motion models. Smooth foot motion, realized through smooth foot *trajectories*, results in decreased wear and tear on motors, gears, and joints and increased motor efficiency. Here, trajectories refer to the time history of position and velocity for the three degrees of freedom of the feet. [CRA86]

To guarantee smooth trajectories for the foot, some constraints on the spatial and temporal qualities between footholds is required. These constraints are realized through the use of curved trajectories. There are many curves with the potential to provide smooth leg motion. The criteria for selecting the two curves used here are:

- maintain an orientation normal to the terrain at liftoff and touchdown, and
- be continuous through the second derivative.

B. RECTANGULAR FOOT MOTION

AquaRobot presently uses a rectangular foot motion. Generally speaking, the feet are moved upward, forward, downward, and backward along straight lines of travel [IWA88a]. Figure 6-1 illustrates the resulting rectangular foot motion generated using the

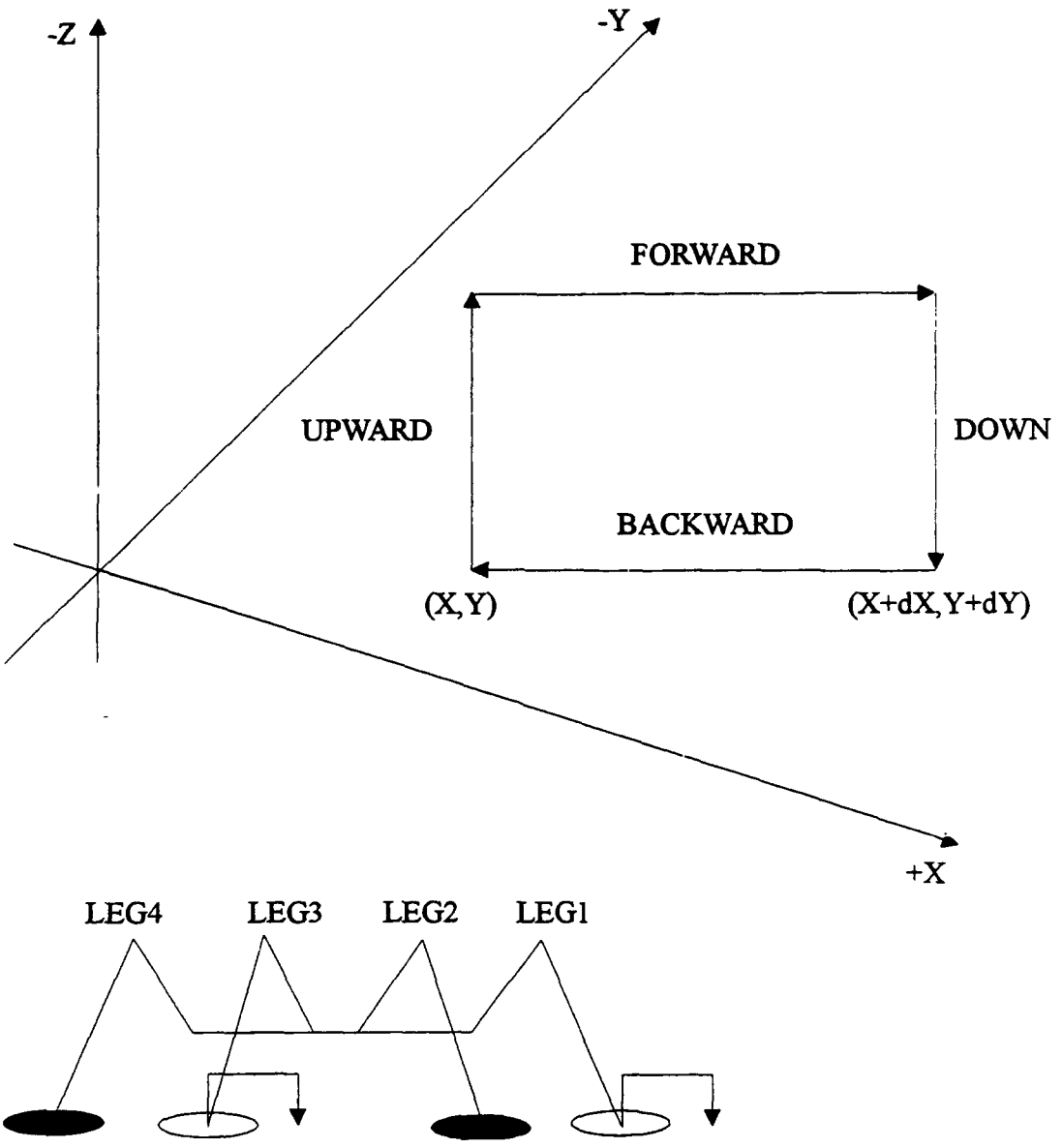


Figure 6-1. Rectangular Foot Motion.

existing algorithm. If (x, y) is an arbitrary foot point, the new foot point is determined using $(x+dx, y+dy)$. The junction of the straight lines forming the rectangle indicate points where motor speed control is discontinuous.

C. SMOOTH FOOT MOTION – ELLIPSE

An elliptical foot motion was chosen as the primary smooth leg motion model to implement. The ellipse meets both criteria established in the introduction to the chapter. Figure 6-2 illustrates the construction and result of an elliptical foot motion trajectory. The parametric representation of the arbitrary foot position (x, y) begins with

$$\begin{aligned} x &= a\cos\theta \\ y &= b\sin\theta. \end{aligned} \tag{6.1}$$

The first derivative of equations 6.1 result in

$$\begin{aligned} \frac{dx}{d\theta} &= -a\sin\theta \\ \frac{dy}{d\theta} &= b\cos\theta. \end{aligned} \tag{6.2}$$

A small segment ds of the ellipse is then calculated using

$$ds = \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}, \tag{6.3}$$

and then

$$\Delta\theta = \frac{\Delta s}{\sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}}. \tag{6.4}$$

The new foot position is then determined using

$$\begin{aligned} x_{new} &= a\cos(\theta + \Delta\theta) \\ y_{new} &= b\sin(\theta + \Delta\theta). \end{aligned} \tag{6.5}$$

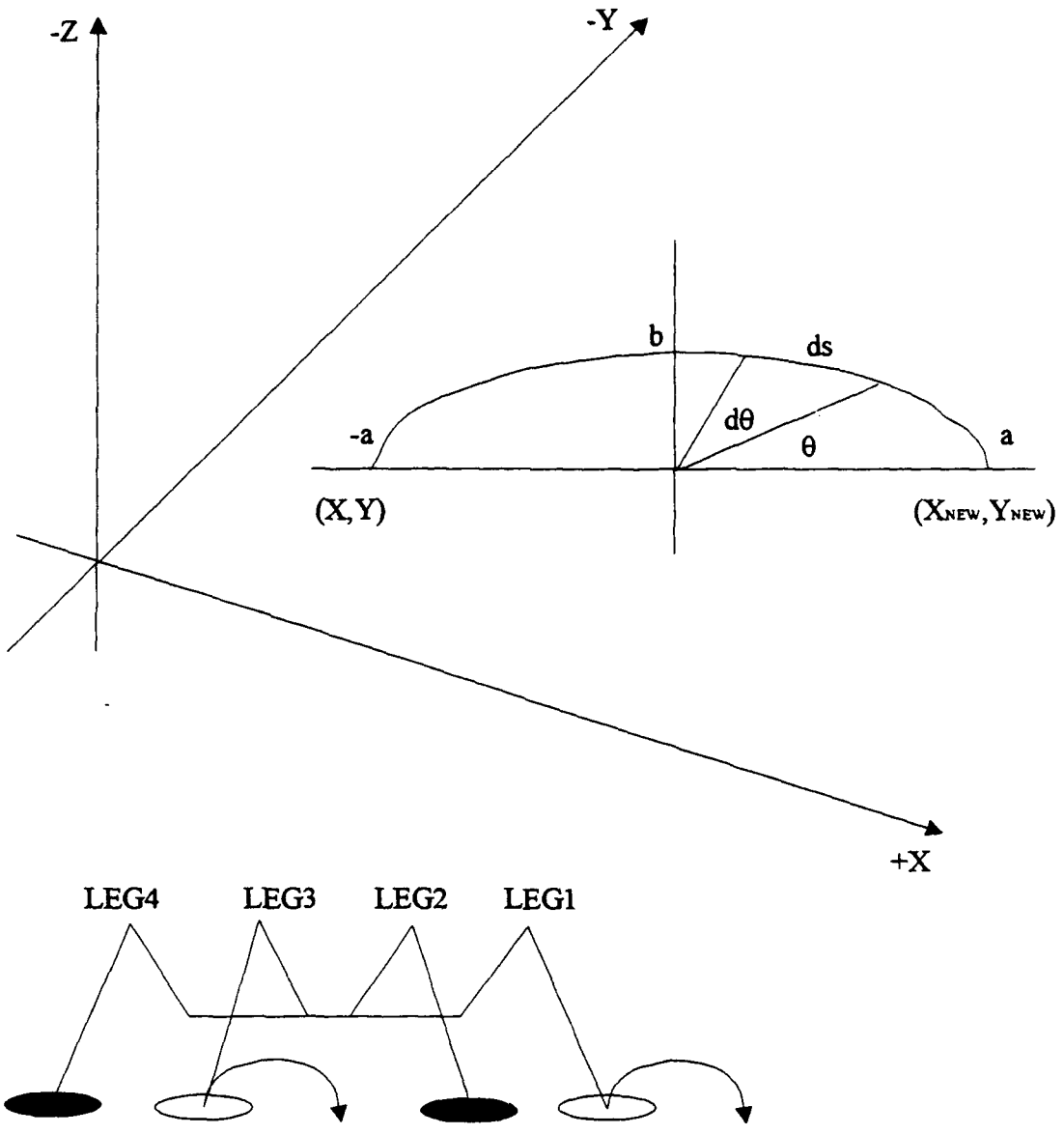


Figure 6-2. Elliptical Foot Motion.

The positive minor axis b of the ellipse is constrained such that the fixed CB height is retained and the joint limits are not exceeded.

D. SMOOTH FOOT MOTION – CYCLOID

A cycloidal foot motion was chosen as the alternate smooth leg motion model to implement. The cycloid meets both criteria established in the introduction to this chapter. Figure 6-3 illustrates the construction and result of a cycloidal foot motion trajectory. The parametric representation of the arbitrary foot position (x, y) begins with

$$\begin{aligned} x &= a(\theta - \sin\theta) \\ y &= b(1 - \cos\theta). \end{aligned} \tag{6.6}$$

The first derivative of equations 6.6 result in

$$\begin{aligned} \frac{dx}{d\theta} &= a(1 - \cos\theta) \\ \frac{dy}{d\theta} &= a\sin\theta. \end{aligned} \tag{6.7}$$

A small segment ds of the cycloid is then calculated using

$$ds = \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}, \tag{6.8}$$

and then

$$\Delta\theta = \frac{\Delta s}{\sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}}. \tag{6.9}$$

The new foot position is then determined using

$$\begin{aligned} x_{new} &= a\left[(\theta + \Delta\theta) - \sin(\theta + \Delta\theta)\right] \\ y_{new} &= a\left[1 - \cos(\theta + \Delta\theta)\right]. \end{aligned} \tag{6.5}$$

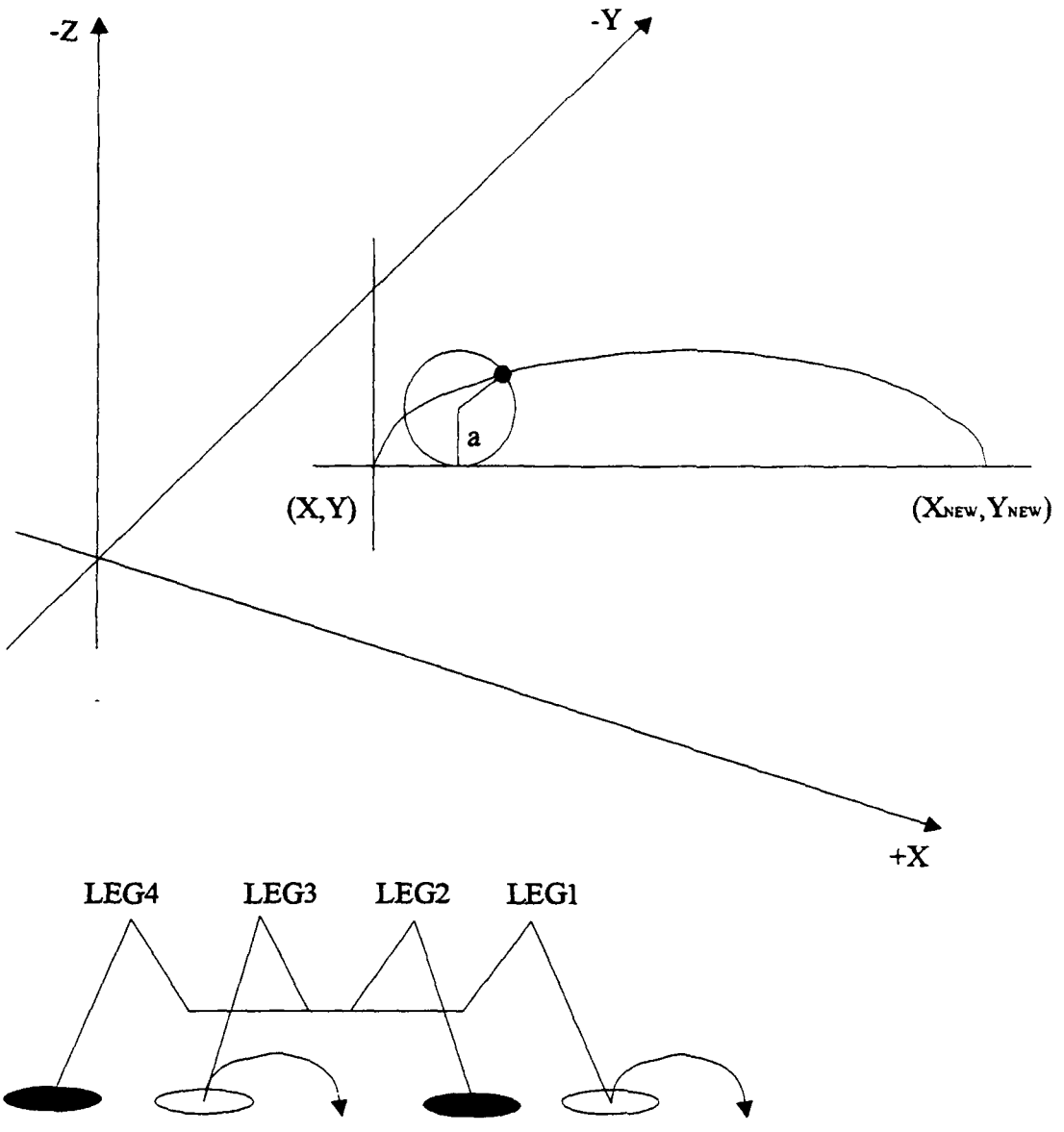


Figure 6-3. Cycloidal Foot Motion.

The radius a of the circle, of which a point on its circumference traces the cycloid, is constrained such that the fixed CB height is retained and the joint limits are not exceeded.

E. SUMMARY

In this chapter, two smooth foot motion models were developed: elliptical and cycloidal. The next chapter describes the gait planning process for the alternating tripod gait used in this study.

VII. GAIT PLANNING

A. INTRODUCTION

"Leg motion planning algorithms are designed to generate the foot trajectory for the support and transfer phases and to choose the desired footholds for the transfer legs based on the body velocity and the constrained working volume (CWV)" [LEE88a]. Leg motion planning is sometimes decomposed into separate *gait selection* and *gait implementation* tasks. Gait selection is generally regarded as the prior decision to use, or select, a specific gait. This study concentrates on the alternating tripod gait because it allows reasonably fast walking, yet maintains static stability at all times. Gait implementation, then, is the execution of the chosen gait through foothold selection and joint angle determination. Because there is so much interaction between the gait selection and gait implementation tasks, we choose to combine them into one task termed *gait planning*.

B. RECTANGULAR TRIPOD GAIT

The fundamental walking algorithm for *AquaRobot* is an up, over, and down motion of a group of three legs. As in previous analyses, we call the first group of legs TRIPOD0 (LEGs 1,3,5) with body-fixed coordinates $(x1, y1, z1)$, $(x3, y3, z3)$, and $(x5, y5, z5)$, respectively. Similarly, we call the second group of legs TRIPOD1 (LEGs 2,4,6) with body-fixed coordinates $(x2, y2, z2)$, $(x4, y4, z4)$, and $(x6, y6, z6)$. We begin with all legs touching the ground. A direction for the robot body to travel is selected. Then the distance for the robot body to move is determined, possibly based on a desired *goal* point. From this information, the new foot positions are chosen. Then, the body-fixed coordinates of a selected group of legs are changed by the difference between the existing foot positions

and the new foot positions. The LEGs in a group are then commanded to move *dx* in the x-axis direction, *dy* in the y-axis direction, and *dz* in the z-axis direction:

$$(x1, y1, z1) = (x1 + dx, y1 + dy, z1 + dz), \tag{7.1}$$

$$(x3, y3, z3) = (x3 + dx, y3 + dy, z3 + dz), \text{ and} \tag{7.2}$$

$$(x5, y5, z5) = (x5 + dx, y5 + dy, z5 + dz). \tag{7.3}$$

TRIPOD0 is generally selected as the first group of legs to move, although this is a purely arbitrary decision. Once the TRIPOD0 feet are free from the ground, the robot body is moved along the desired direction of travel. When the TRIPOD0 feet are firmly planted, calculations are begun to move TRIPOD1, using the commanded motion derived from:

$$(x2, y2, z2) = (x2 + dx, y2 + dy, z2 + dz), \tag{7.4}$$

$$(x4, y4, z4) = (x4 + dx, y4 + dy, z4 + dz), \text{ and} \tag{7.5}$$

$$(x6, y6, z6) = (x6 + dx, y6 + dy, z6 + dz). \tag{7.6}$$

Alternating the groups of legs, while choosing the same direction, causes *AquaRobot* to travel along a straight line with the body height held constant. [IWA88a]

We choose to call this gait the *Rectangular Alternating Tripod Gait* (RATG) because the free leg trajectories describe a rectangular pattern. Figure 7-1 illustrates a representative leg motion for the RATG. First, the three legs of the free tripod (in this case, TRIPOD0) are lifted perpendicular to the terrain until KNEE joint limits are reached. Essentially, the free legs are lifted straight upward until the feet are at their maximum height. Next, the body is moved in the direction of travel until HIP joint limits are reached. Then the free legs are moved in the direction of travel, horizontal with the terrain, until the free feet are over the desired touchdown positions. Finally, the free legs are lowered

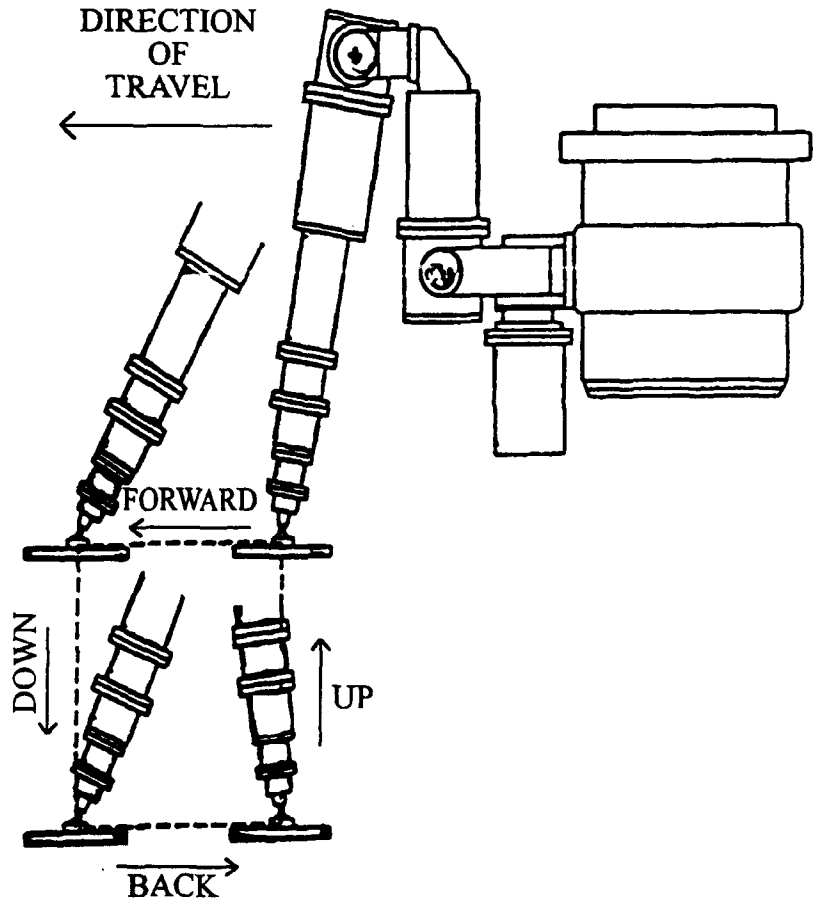


Figure 7-1. Rectangular Tripod Gait Leg Motion.

perpendicular to the terrain until contact sensors in each foot indicate that the leg motion is completed.

Gaits can be expressed as functions of distance or time in *gait diagrams* [TOD86]. Figure 7-2 illustrates the distance traveled at ten successive time intervals, $t0$ through $t9$, when a RATG is used. A typical sequence of events for the RATG is:

- body and leg components in START posture, ($t0$),
- free and lift the legs of TRIPOD0 (LEGs 1, 3, 5), ($t1$),
- move the CB in the direction of travel, ($t2$),
- move the free legs of TRIPOD0 in the direction of travel, ($t3$),
- place the legs of TRIPOD0, ($t4$),
- free and lift the legs of TRIPOD1 (LEGs 2, 4, 6), ($t5$),
- move the CB in the direction of travel, ($t6$),
- move the free legs of TRIPOD1 in the direction of travel, ($t7$),
- place the legs of TRIPOD1, ($t8$), and then the process repeats, ($t9$).

In Figure 7-2, a solid dot (•) indicates the foot is touching the terrain and an empty dot (◦) indicates the foot is free from the terrain. A cross (+) indicates the position of the robot's CB.

Figure 7-3 shows the RATG as a function of time. The motion of each major body component (TRIPOD0, CB, TRIPOD1) is illustrated using a timing diagram. In Figure 7-3, a solid bar (–) along the upper line represents movement of the associated body component with respect to the ground. A solid bar along the lower line indicates no movement of that particular body component with respect to the ground. Analysis of Figure 7-3 reveals several times when body components are not moving. For example, during time $t1$ neither the CB nor TRIPOD1 are moving. These areas indicate times when joint motors are not powered. Additionally, the CB moves during only one time interval

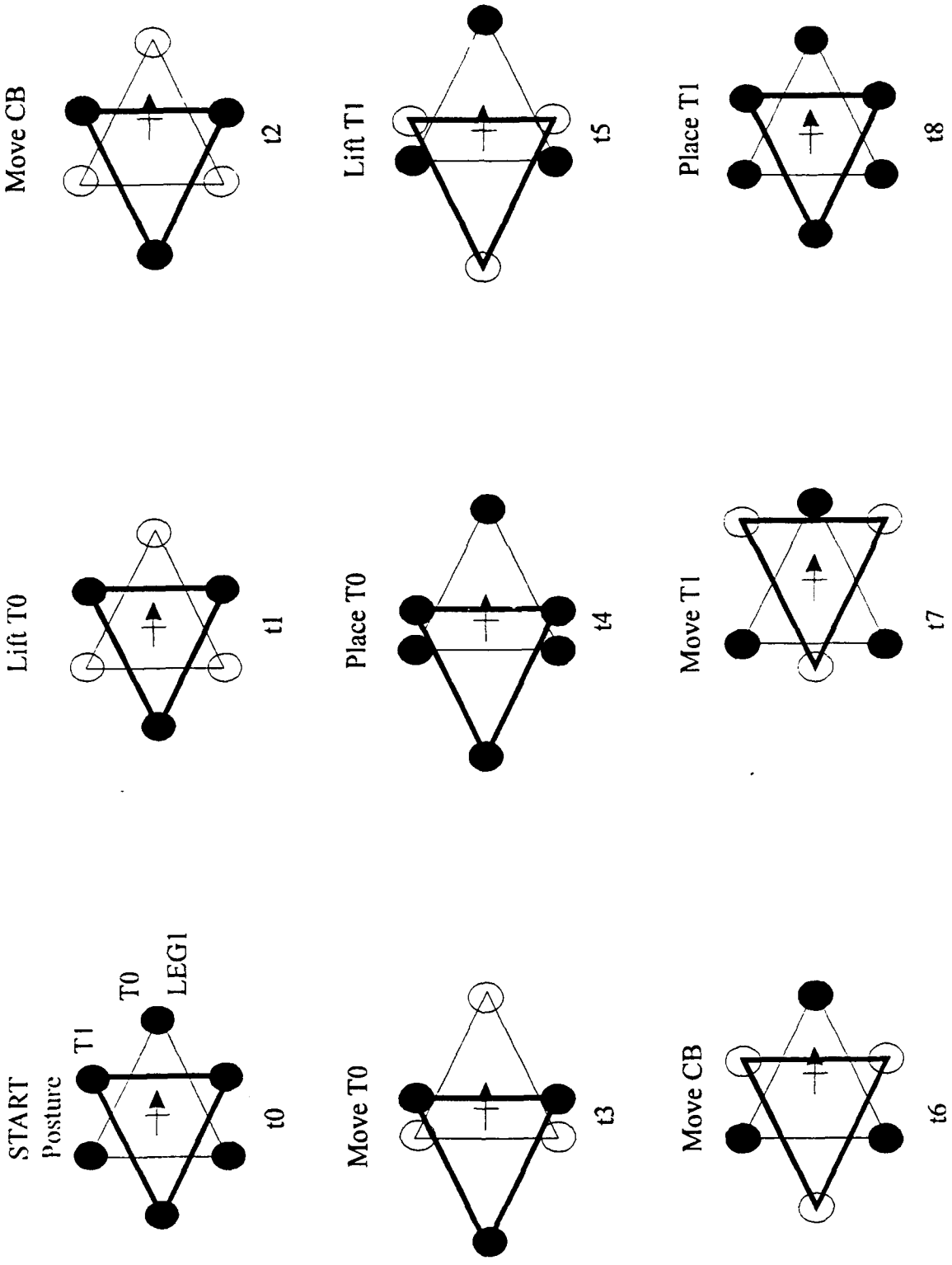


Figure 7-2. Rectangular Gait as a Function of Distance.

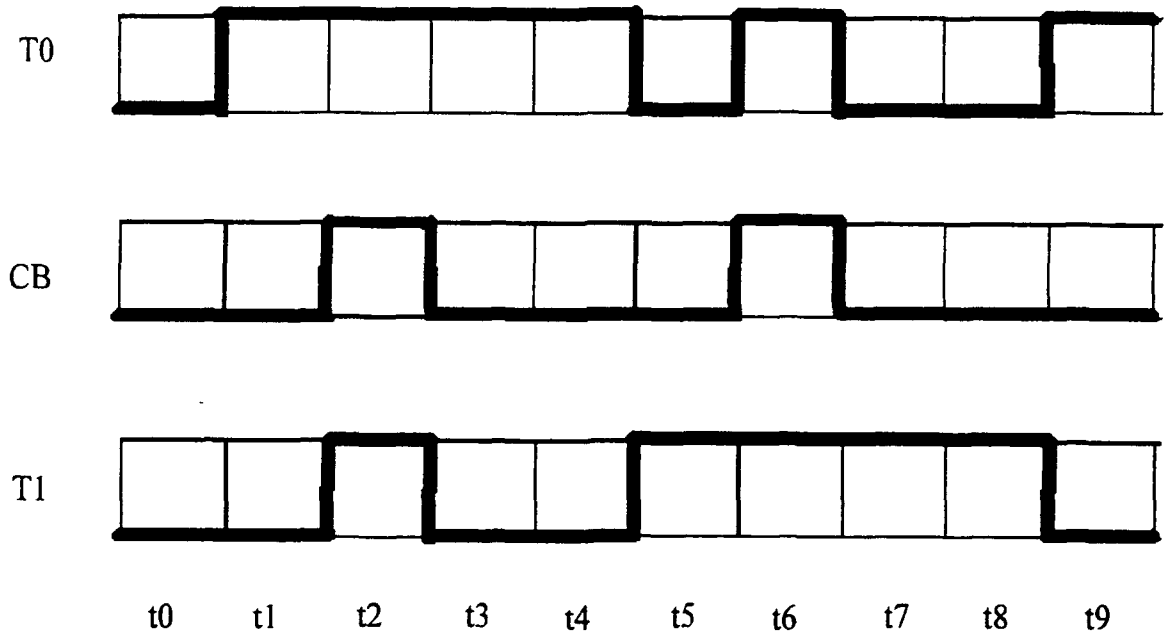


Figure 7-3. Rectangular Gait as a Function of Time.

($t2$ or $t6$) for each TRIPOD cycle. These periods of discontinuous motion are inefficient, resulting in jerky body movement and slower over-the-ground speed.

C. DISCRETE TRIPOD GAIT

The *Discrete Alternating Tripod Gait* (DATG) was developed as the natural next step after the rectangular gait. For our purposes, non-continuous motion of the CB is known as *discrete* motion. The DATG provides for discrete motion of *AquaRobot's* CB. The distance the CB travels is constrained by workspace and stability limits. For ease of calculation, the three feet in a tripod describe a regular triangle. This approach does not necessarily lead to the maximum possible stride or the minimum time to cover a distance. Nevertheless, stability is easily assured. The DATG includes an elliptical foot motion trajectory as a means of allowing continuous foot motion between ground contact points. A typical sequence of events for the DATG is:

- body and leg components in START posture, ($t0$),
- plan an elliptical path for and move TRIPOD0 to its destination, ($t1$),
- move the CB in the direction of travel until a limit is reached, ($t2$),
- plan an elliptical path for and move TRIPOD1 to its destination, ($t3$),
- move the CB in the direction of travel until a limit is reached, ($t4$), and
- continue repeating the process, ($t5$ through $t9$).

Figure 7-4 shows the DATG as a function of distance. Figure 7-5 shows the DATG as a function of time. The major difference between the rectangular and discrete gaits is the leg motion. Except for periods when the feet are leaving or contacting the terrain, or when only the CB is moving, the feet are in continuous motion. The leg joint motors for the free legs in the DATG, then, are operating almost continuously. However, the CB motion is discontinuous in both gaits.

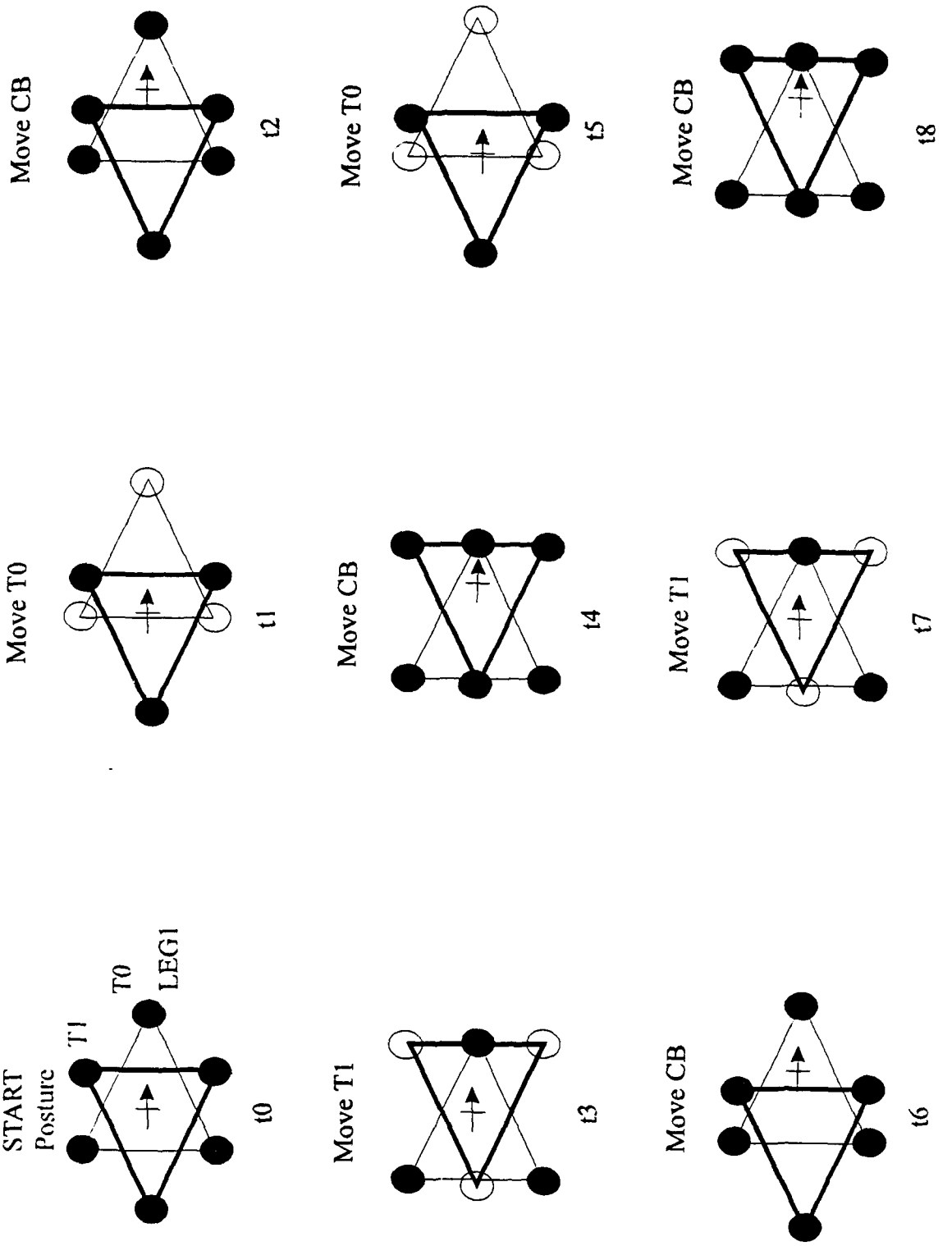


Figure 7-4. Discrete Gait as a Function of Distance.

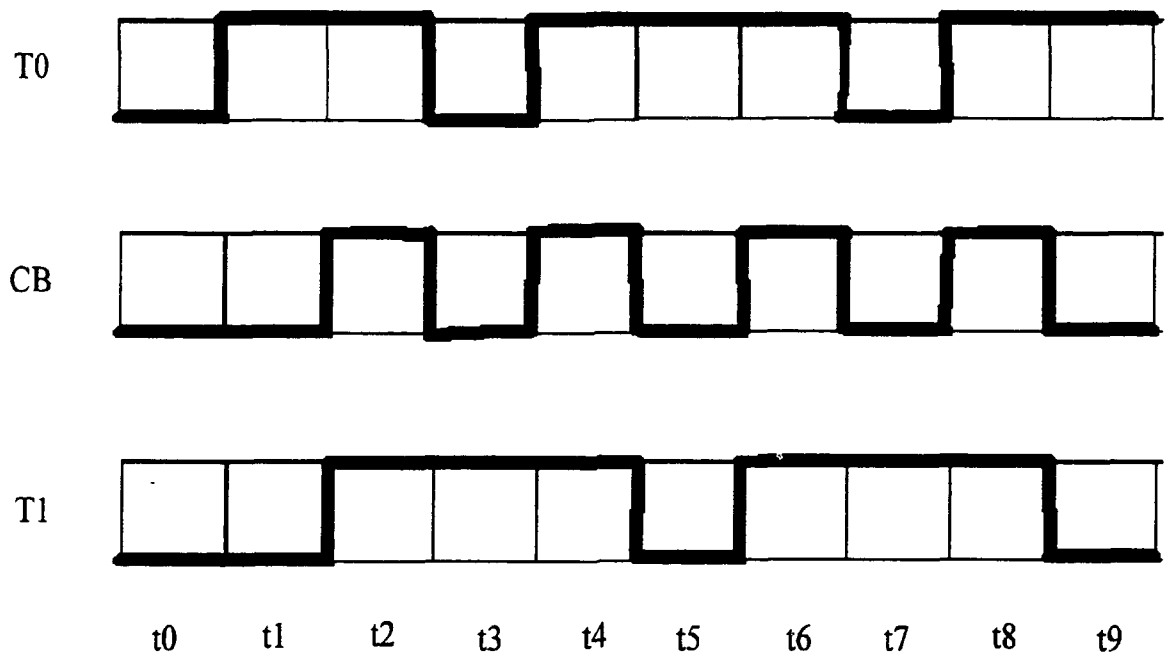


Figure 7-5. Discrete Gait as a Function of Time.

D. CONTINUOUS TRIPOD GAIT

The *Continuous Alternating Tripod Gait* (CATG) was designed to allow *continuous* motion of the CB, independent of the grouped leg (tripod) motions. The sequence of events is similar to the DATG, except the CB moves continuously. TRIPODs zero (0) and one (1) still alternate and the foot trajectories are still elliptical; however, the CB is not allowed to halt once the walking begins (unless the goal point is reached). The supporting tripod provides the motion for the CB.

The CB motion is synchronized with the motion of the tripods. The desired foot positions are used to generate an elliptical trajectory for the selected free tripod in the transfer phase. Then the distance between the existing and desired foot positions is divided into small segments. The length of these segments is then used to determine the motion the CB makes in the direction of travel in the *xy* plane. The body height is held constant, so the CB is not moved in the *z* plane. For every increment of free tripod foot motion in the *xy* plane, the supporting tripod increments the CB motion so the CB completes one-half the distance the free tripod travels in the same time period. The CB moves one-half the foot distance because of workspace restrictions. Actually, the supporting tripod joints are moving, thereby moving the CB indirectly. The increments are chosen arbitrarily small, in our case 100, so the CB appears to move continuously. The CB moves in the direction of travel during each tripod supporting phase and when all six legs are on the ground.

From observation of the CATG, the maximum possible stride is 146.04 centimeters. Therefore, the maximum possible CB motion is 73.02 centimeters. The *AquaRobot* simulation can be run with a default maximum stride value slightly less than 146 centimeters to compensate for the edges of the footpads touching at the extreme limits of the workspace.

A typical sequence of events for the CATG follows. No time steps are given because all major body components are continuously in motion. Initially, the body and leg components are in the START posture. The desired foot positions for each foot in one tripod are determined using workspace and stability constraints. The stride chosen is the minimum distance any one of the three feet in a tripod grouping can move in its individual workspace. TRIPOD0 is arbitrarily selected and an elliptical path is planned to move its feet one increment towards their new destination. For the first step, the CB is moved a distance equal to the maximum stride. Therefore, then new location for the CB to travel to is calculated using

$$CB(x_w) = CB(x_w) + (stride) * \cos(direction), \text{ and} \tag{7.7}$$

$$CB(y_w) = CB(y_w) + (stride) * \sin(direction). \tag{7.8}$$

TRIPOD1 then moves the CB the distance required to attain the new CB coordinates. For the next steps, the CB is moved a distance equal to one-half the maximum stride. In this case, the new location for the CB to travel to is calculated using

$$CB(x_w) = CB(x_w) + \left(\frac{stride}{2}\right) * \cos(direction), \text{ and} \tag{7.9}$$

$$CB(y_w) = CB(y_w) + \left(\frac{stride}{2}\right) * \sin(direction). \tag{7.10}$$

After each incrementation of the CB position, its location is compared to the goal point. If the distance from the CB to the goal point is less than would result if the maximum stride were used, the lessor distance is used. Then, the new stride length becomes twice the distance from the CB to the goal point.

Figure 7-6 shows the CATG as a function of distance. The motion of the legs is the same as the DATG. Figure 7-7 shows the CATG as a function of time. This figure illustrates the continuous motion of the major body components.

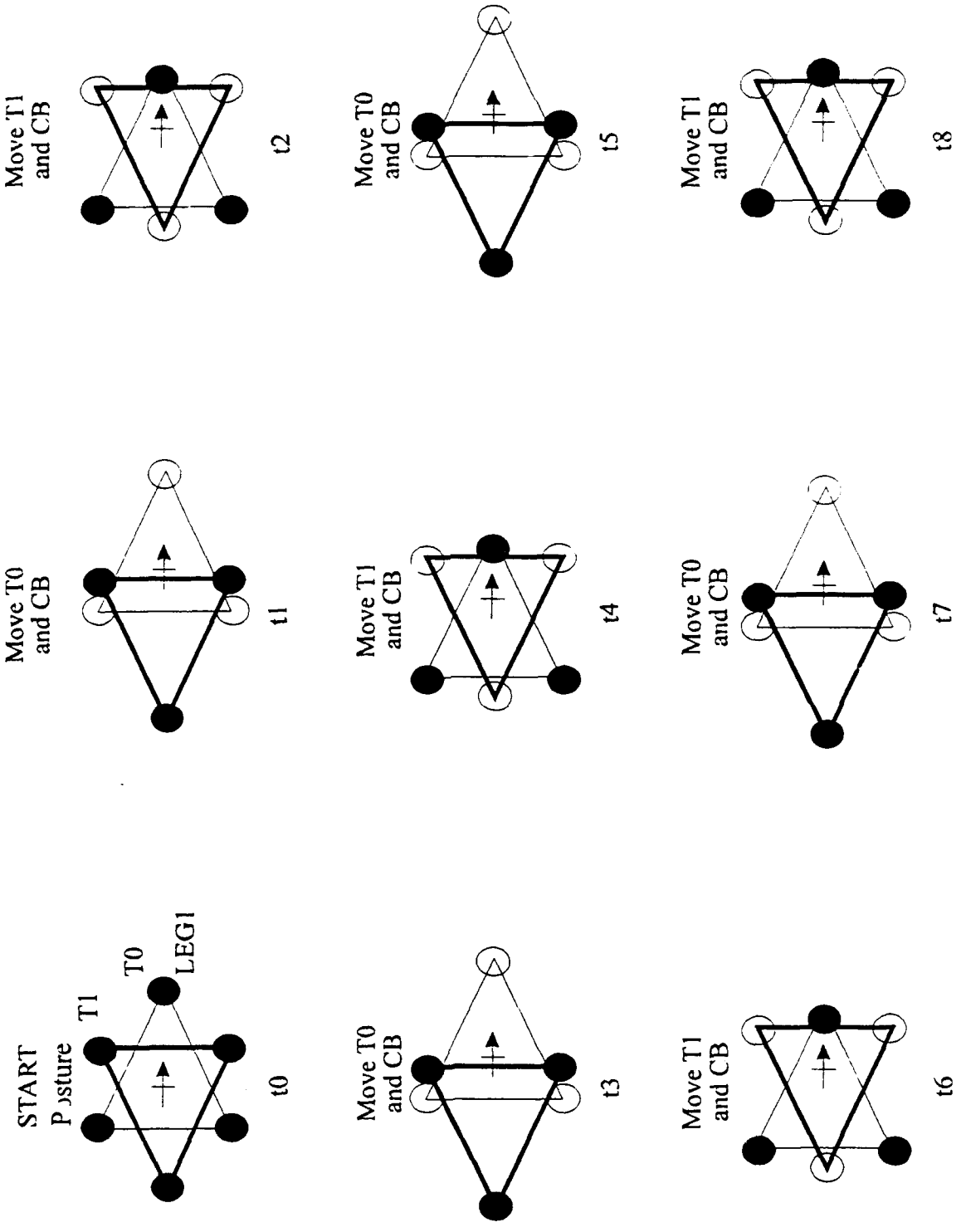


Figure 7-6. Continuous Gait as a Function of Distance.

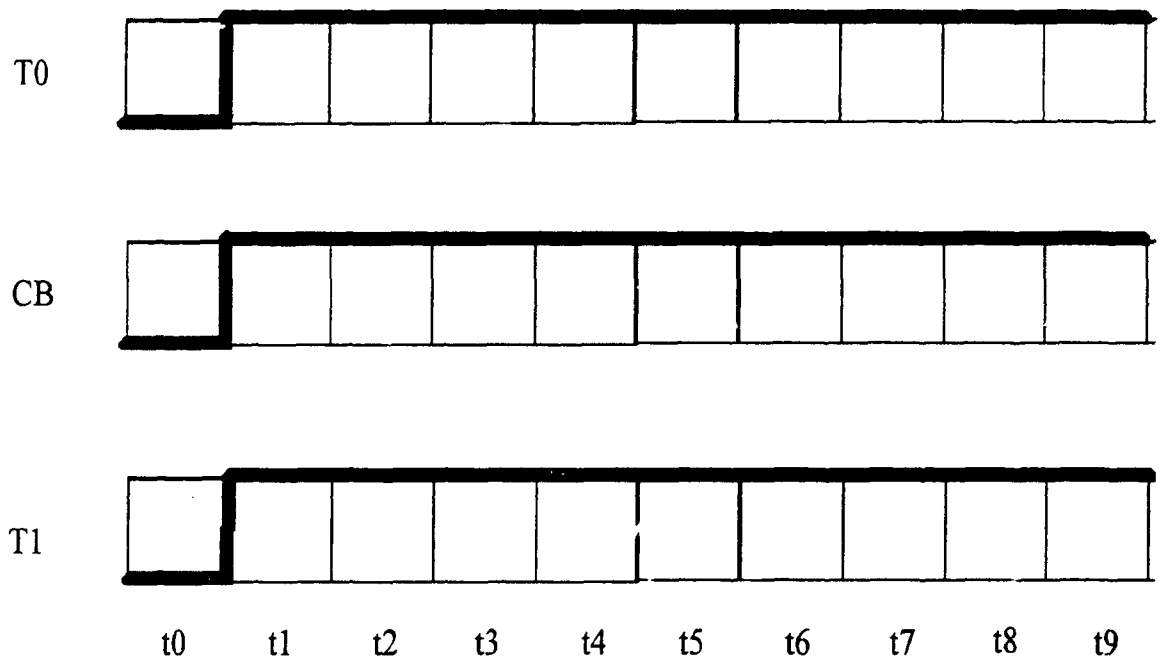


Figure 7-7. Continuous Gait as a Function of Time.

E. SUMMARY

In this chapter, the gait planning algorithms for the rectangular, discrete, and continuous alternating tripod gaits were described. The next chapter implements the discrete and continuous gait planning algorithms in computer code.

VIII. *AQUAROBOT* SIMULATOR

A. INTRODUCTION

In this chapter, the *AquaRobot* Simulator program is presented. Initially, the simulation facilities, tools, and methods are described. Then the internal structure and flow of the program is discussed. One part of the program is a three-dimensional (3D) stick-figure simulation of the DATG. Another part of the program is a 3D stick-figure simulation of the CATG. A complete listing of program code is found in Appendix C. A separable User's Guide, with instructions on how to use the program, is included as Appendix D.

B. SIMULATION FACILITIES

A critical part of this thesis included integrating the gait planning module design with a previously developed graphics simulation program [DAV93]. As a result, some graphics code required revision or augmentation. Additionally, the gait planning code modules required some transportability between the *AquaRobot* control computer (NEC 486-based Personal Computer with MS-DOS operating system) and the *AquaRobot* Simulator (Silicon Graphics Personal Iris Workstation). To ensure some level of transportability between the personal computer and the workstation, the ANSI standard C format was adopted. The graphics code modules are specific to the Silicon Graphics Workstation.

Many portions of the gait planning algorithms were originally written and tested in Matlab, a "C"-based development language. Appendix B contains the Matlab code. Once the basic algorithms were verified, they were converted to the C++ language for implementation in the *AquaRobot* Simulator. The C++ language was used for graphics, for

interfacing between the gait planning and graphics modules, and whenever specific objects (legs, body, etc.) could be defined. Appendix C contains C++ code.

C. MODULE STRUCTURE

Figure 8-1 contains a flow chart of the *AquaRobot* Simulator program. The simulation is essentially one program with options for either the DATG or the CATG. Typing *aqua* starts the simulation. The user then determines the distance, and thereby the direction, the simulated *AquaRobot* travels at the outset of the program: the goal point, in (x,y) coordinates, is entered from the keyboard. Any arbitrary distance (direction) is acceptable. Once the graphics portion of the program begins, the user can select the camera viewing angle. When the simulated *AquaRobot* reaches the desired goal point, it stops. A description of each functional module follows.

The simulator files are separated into four modules. The files that comprise the Makefile, Graphics, and Matrix Manipulation Modules are only cursorily addressed here. The Gait Planning Module is described in more detail.

1. Makefile

The *Makefile* allows the *make* utility to intelligently compile the program. This file includes instructions for how and when to compile the various files that comprise the *AquaRobot* Simulator. The *AquaRobot* simulation program uses the UNIX *make* utility to link together the 27 individual files with the graphics, math, and standard input-output libraries. In this case, the Makefile generates the executable program *aqua*.

2. Graphics

The 3D stick-figure graphics code was originally written by Sandra Davidson [DAV93]. The graphics files (code modules) are included in Appendix C for easier reference. Some file names were changed to avoid software configuration management

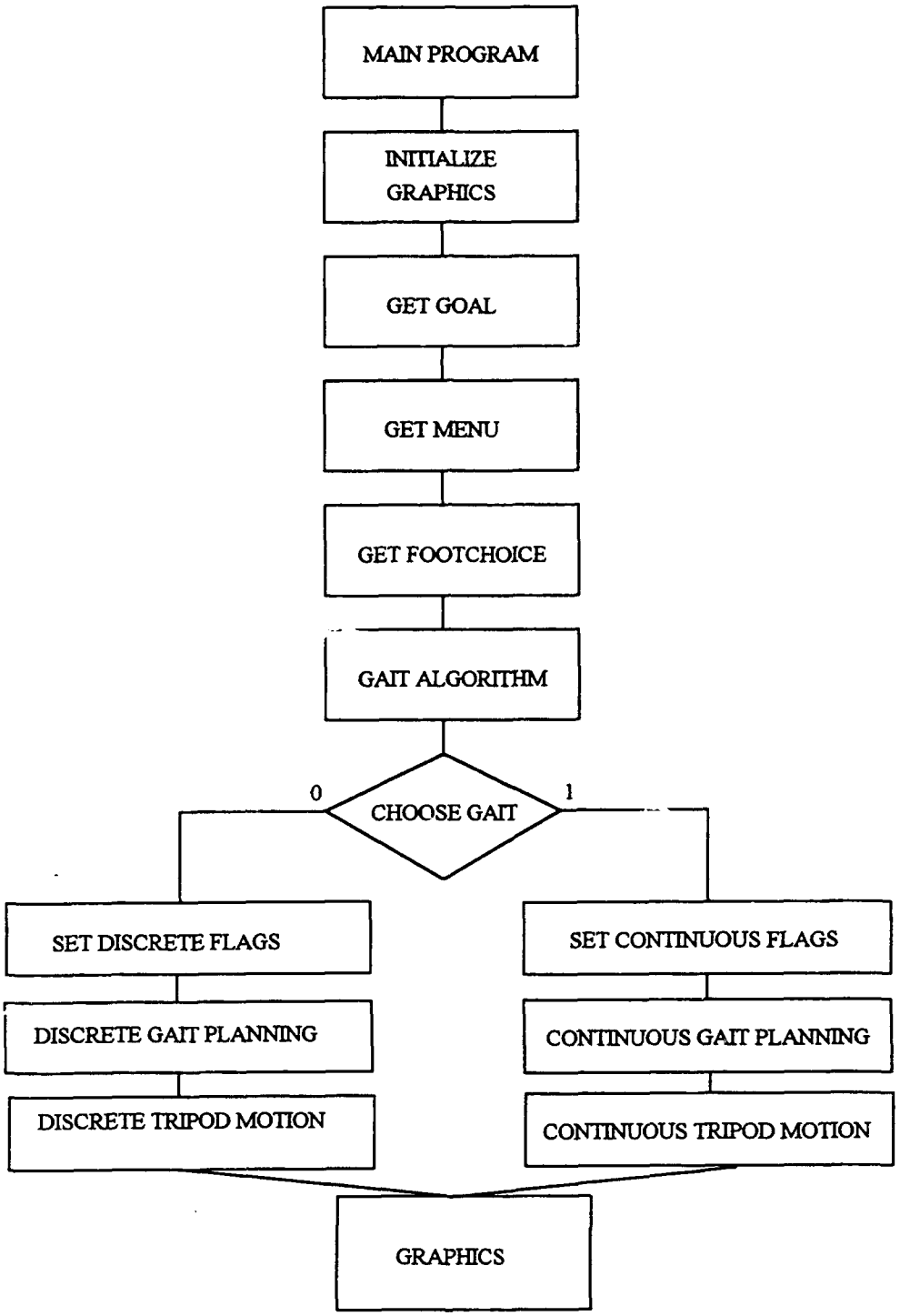


Figure 8-1. *AquaRobot* Simulator Flow Chart.

problems. Additionally, some graphics code was modified because of the requirements of the gait planning code. For example, the FARCLIPPING value and the CAMERA viewing angles were changed in the *bot.H* file and the CB height was changed in the *AbBody.C* file. Because the Graphics Module polls the Gait Planning Module, a suitable interface between the two was constructed in the *Kinematics.H* and *Kinematics.C* files. The graphics code consists of the following files:

- *AbBody.H* and *AbBody.C*;
- *AbLeg.H* and *AbLeg.C*;
- *AbRigid.H* and *AbRigid.C*;
- *bot.H* and *bot.C*;
- *Kinematics.H* and *Kinematics.C*;
- *Link.H* and *Link.C*;
- *Link0.H* and *Link0.C*;
- *Link1.H* and *Link1.C*;
- *Link2.H* and *Link2.C*; and
- *Link3.H* and *Link3.C*.

3. Matrix Manipulation

The matrix manipulation code was also originally written by Sandra Davidson [DAV93]. It provides 4 x 4 matrix multiplication capability. This code is an essential part of the graphics code, although it is not used in the gait planning code. The Matrix Manipulation Module includes the following files, found in Appendix C:

- *MatrixMy.H* and *MatrixMy.C*.

4. Gait Planning

The Gait Planning Module (GPM) consists of the following files:

- *arconst.H;*
- *arfunc.H* and *arfunc.C*; and
- *artpgait.H* and *artpgait.C*.

To support easier reference and maintenance, most constants are grouped together into the file *arconst.H* (**aquarobot constants**).

The *arfunc.H/C* (**aquarobot functions**) files contain 13 functions. The *min* and *max* functions simply return the minimum or maximum of two expressions, respectively. The *ellipse* function calculates the incremental foot trajectory along an elliptical path between the current foot position and the desired foot position. The *kinematics* function performs kinematics for the Gait Planning Module. Kinematics for the Graphics Module are performed in the Graphics *Kinematics.H/C* files. The *inv_kinematics* function performs the inverse kinematics operations. The *world_body* function provides coordinate transformation from the body-fixed coordinate system to the world coordinate system. The *body_world* function provides coordinate transformation from the world coordinate system to the body-fixed coordinate system.

The *maxdistance_25cm* function determines the maximum stride for the workspace defined by the 25 centimeter footpad. The *maxdistance_45cm* function determines the maximum stride for the workspace defined by the 45 centimeter footpad. The *arcint* function calculates the intersection of a directed line and an arc segment. The *segint* function calculates the intersection of a directed line and a line segment. The *stable* function calculates the stability of a tripod. The *staparam* function determines the longitudinal stability margin and the x and y intercepts of the directional ray originating at the CB and terminating at the point of intersection on the selected tripod.

The *artpgait.H/C* (**aquarobot tripod gait**) files 15 functions. The *get_goal* function allows the user to input the desired goal point to which the *AquaRobot* simulation walks. The *get_menu* function allows the user to choose between the DATG or the CATG. The *get_footchoice* function allows the user to determine whether the 25 centimeter or 45 centimeter footpad is used. The *gait_algorithm* function executes either the DATG or the CATG gait algorithms, as previously determined by the user. The *robot_model* function initializes the joint angles for the simulation.

The *disc_set_flag* function initializes the flags used during the DATG algorithm. The *disc_gait_planning* function determines the walking parameters, such as stride length and direction, used during the DATG algorithm. The *disc_tripod_motion* function calculates and executes incremental joint motions necessary to move the *AquaRobot* simulation using the DATG.

The *cont_set_flag* function initializes the flags used during the CATG algorithm. The *cont_gait_planning* function determines the walking parameters, such as stride length and direction, used during the CATG algorithm. The *cont_tripod_motion* function calculates and executes the incremental joint motions necessary to move the *AquaRobot* simulation using the CATG.

Some functions in the *artpgait.C* file are used only for printing out the status of joints, feet, etc. These functions include *print_status*, *print_walkpara*, *print_gnu_data*, and *print_joint_data*.

D. SUMMARY

This chapter described the *AquaRobot* simulation program. A discussion of the simulation facilities, tools, and methods followed. Then, the organization of the program was described. The final chapter contains results and concluding remarks.

IX. CONCLUSIONS

A. RESULTS

Both the DATG and CATG algorithms work successfully. Most of the coding is shared between the two algorithms. The Gait Planning Module, of which the DATG and CATG are a part, requires approximately 1600 total lines of C++ code. Graphics code is excluded from the line count.

Figures 9-1(a), 9-1(b), and 9-1(c) juxtapose the diagrams for the rectangular, discrete, and continuous alternating tripod gaits. It is obvious from the figures that there is a progression of improvement in continuous CB motion fro the RATG to the CATG.

Many areas of the RATG and DATG gait diagrams show no major body component (TRIPOD0, CB, TRIPOD1) motion. These areas indicate times when joint motors are operating discontinuously, i.e., motors are stopped. Starting and stopping the joint motors results in rough, jagged motion. This, in turn, tends to cause abnormal wear of mechanical mechanisms, vibrations resulting from excited mechanical resonances, and inefficiencies of motor operation. Smoother, more continuous motion results in decreased wear and tear on motors, gears, and joints and increased motor efficiency. The CATG, with elliptical foot trajectory, allows these benefits to occur.

Figures 9-2 through 9-7 present the motion of the individual leg joints (HIP, KNEE1, and KNEE2) for the first few tripod cycles of the DATG and the CATG when walking with LEG1 along the *x*-axis. The upper plot in each figure (a) shows the DATG; the lower plot in each figure (b) shows the CATG. As seen in Figures 9-2 and 9-5, there is no HIP motion for the legs of the tripod that lie on the x-axis: LEG1 in TRIPOD0 and LEG4 in

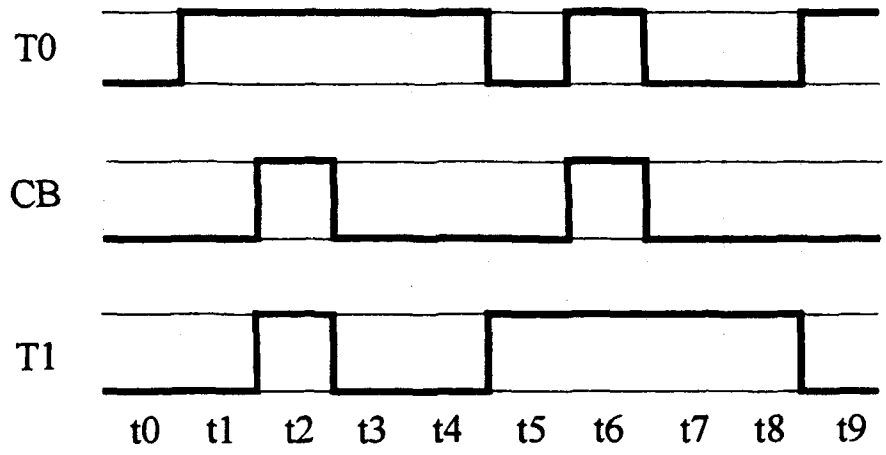


Figure 9-1(a). Rectangular Alternating Tripod Gait.

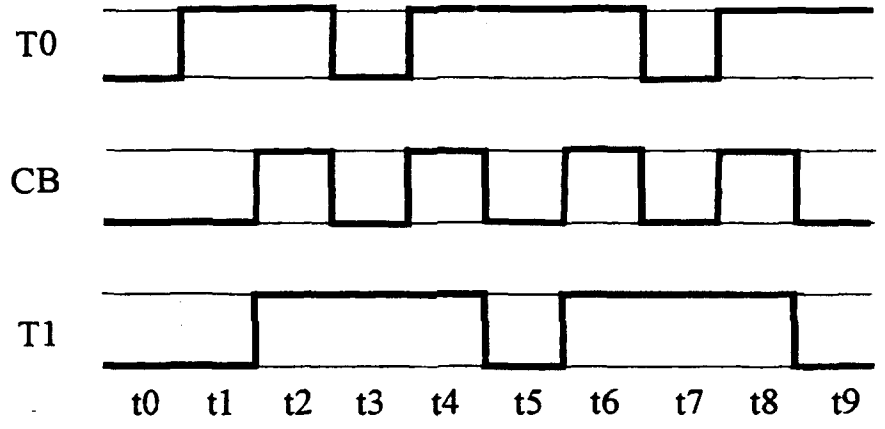


Figure 9-1(b). Discrete Alternating Tripod Gait.

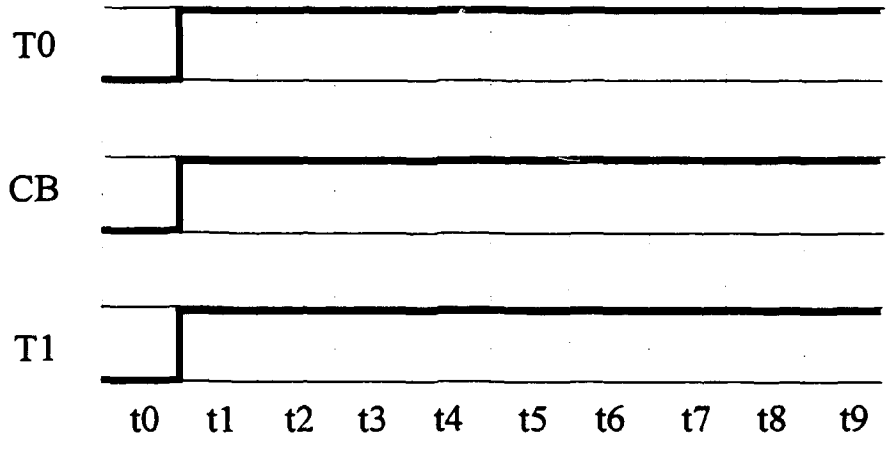


Figure 9-1(c). Continuous Alternating Tripod Gait.

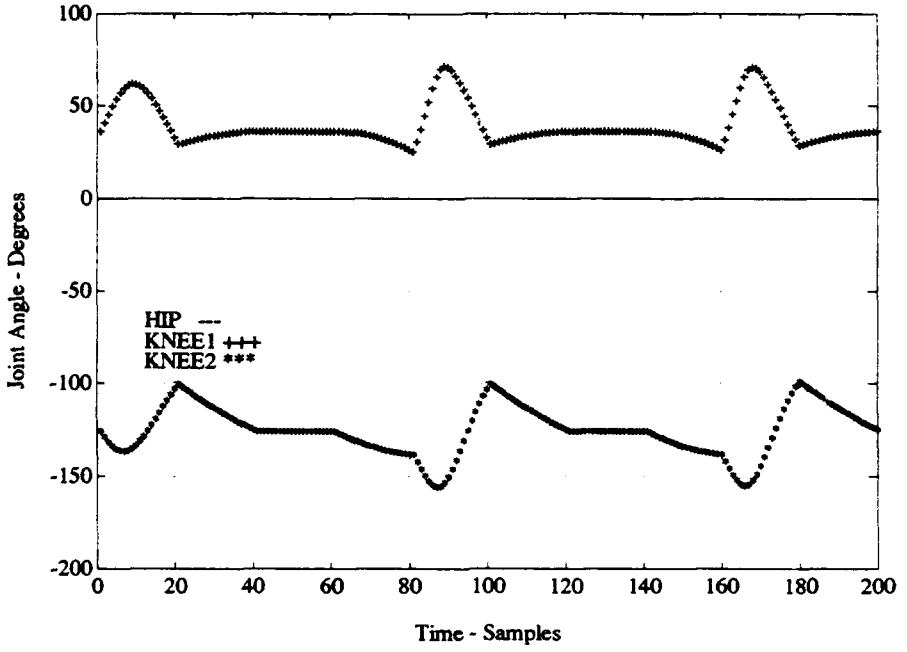


Figure 9-2(a). LEG1 Joint Angle Motion for DATG.

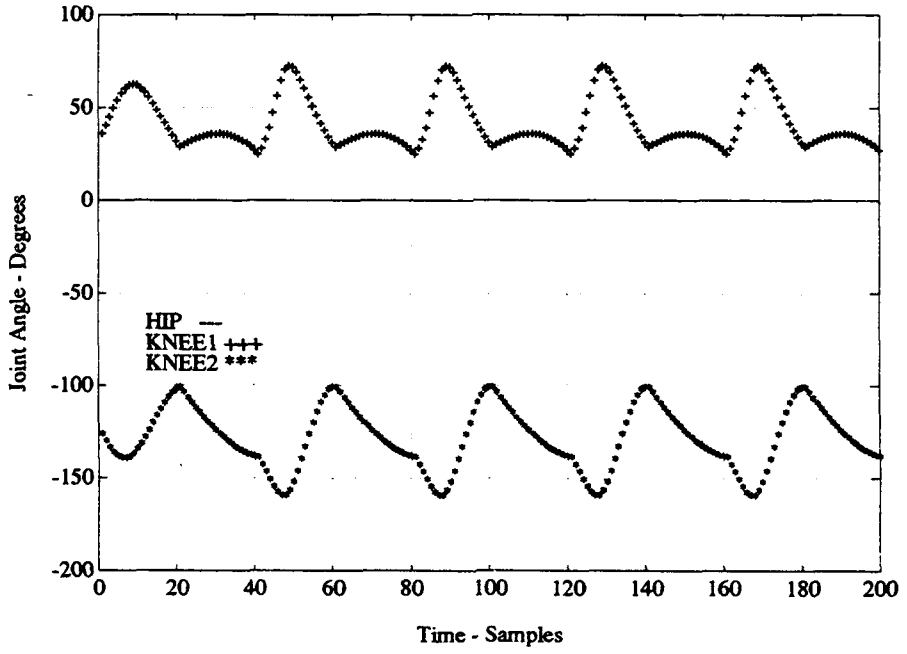


Figure 9-2(b). LEG1 Joint Angle Motion for CATG.

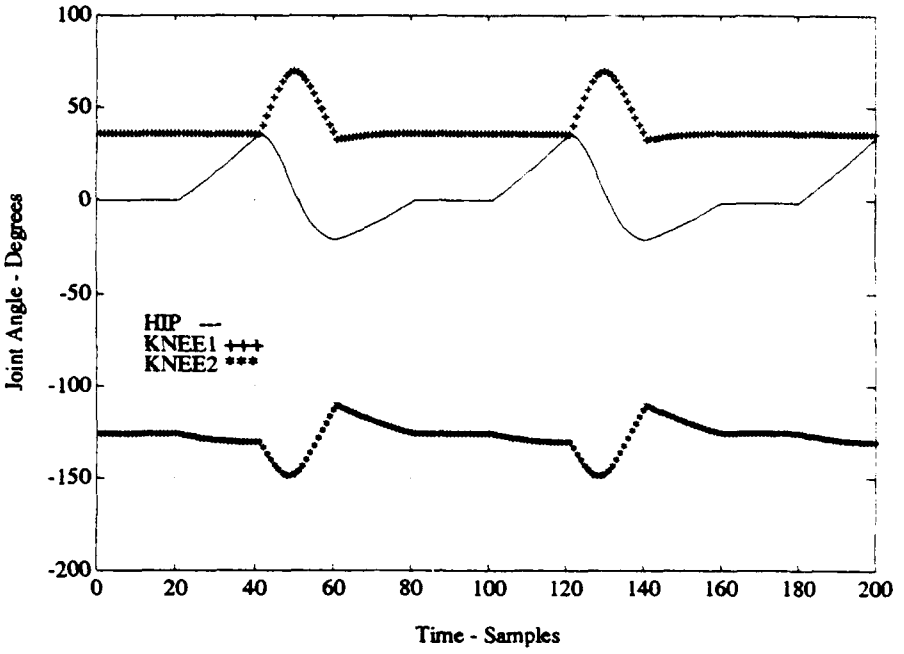


Figure 9-3(a). LEG2 Joint Angle Motion for DATG.

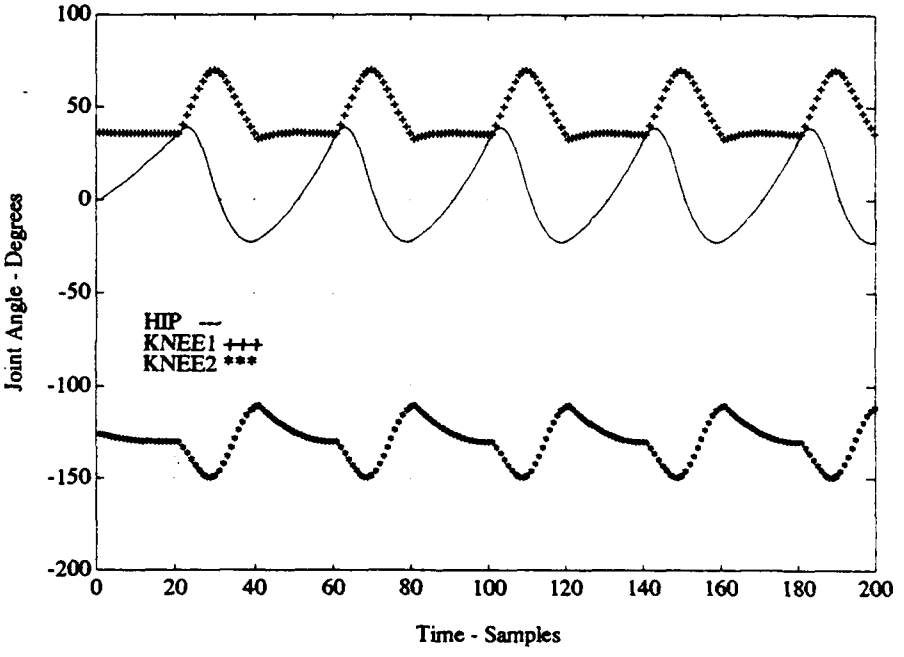


Figure 9-3(b). LEG2 Joint Angle Motion for CATG.

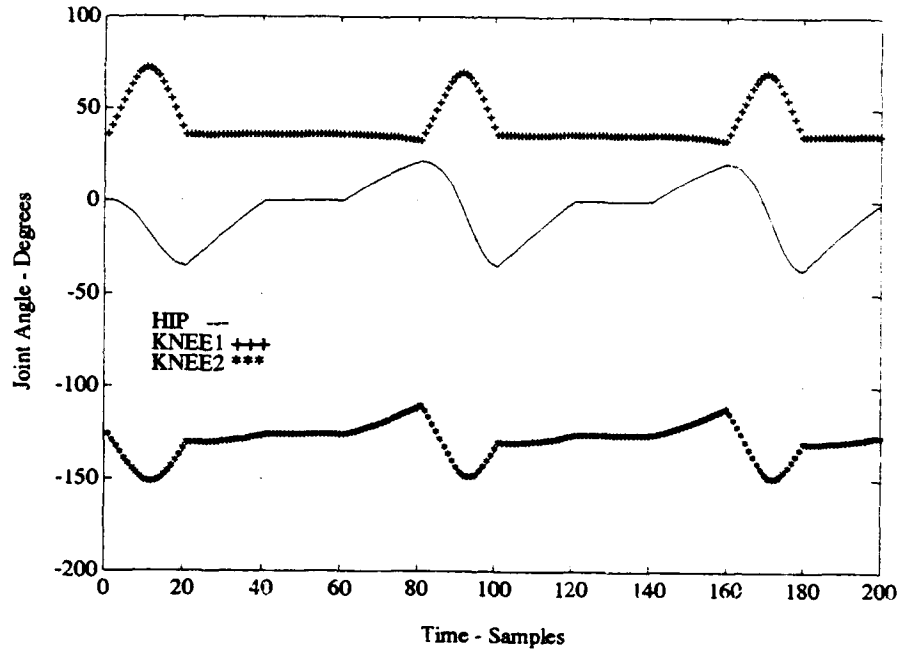


Figure 9-4(a). LEG3 Joint Angle Motion for DATG.

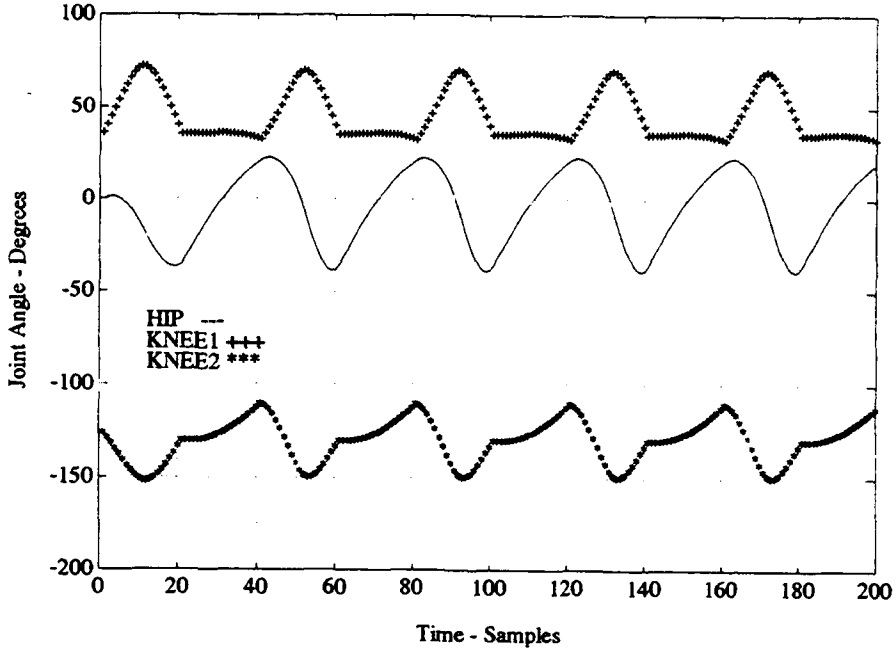


Figure 9-4(b). LEG3 Joint Angle Motion for CATG.

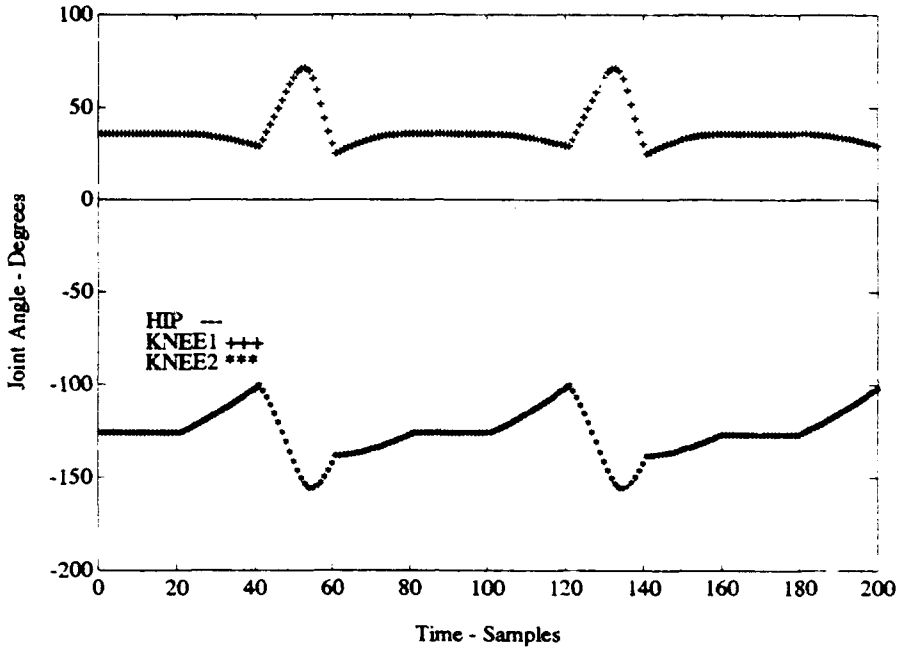


Figure 9-5(a). LEG4 Joint Angle Motion for DATG.

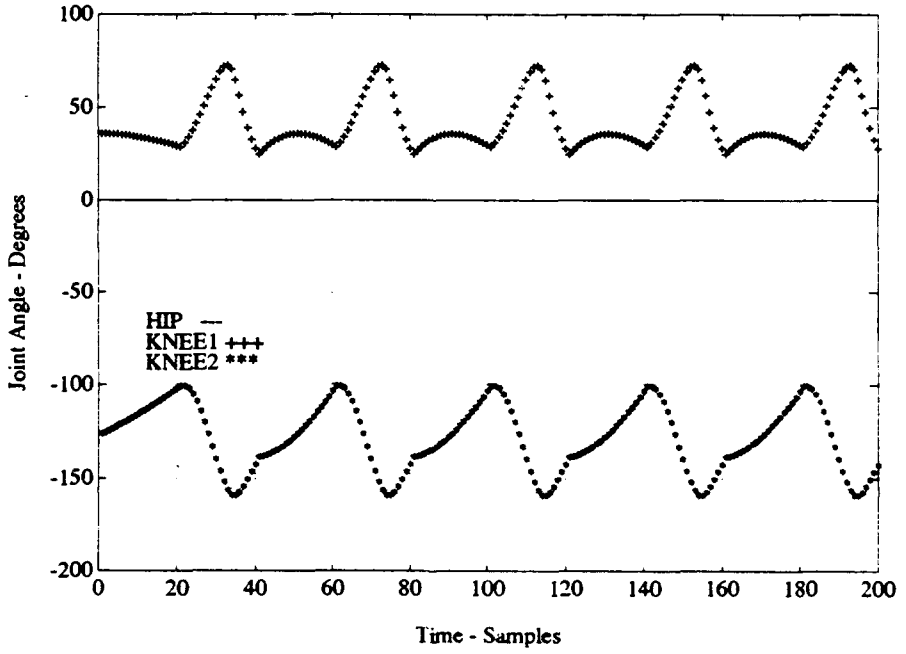


Figure 9-5(b). LEG4 Joint Angle Motion for CATG.

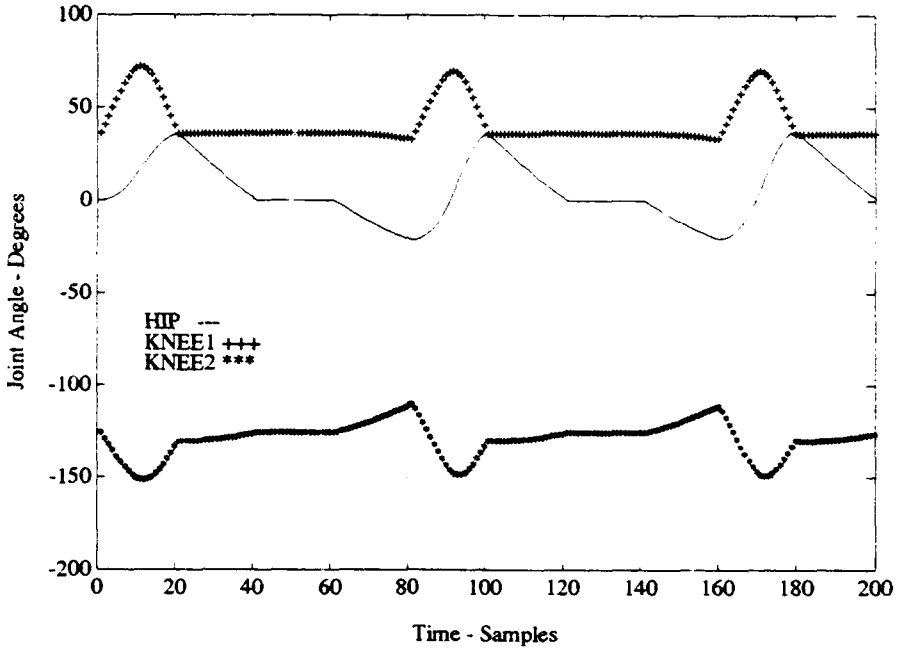


Figure 9-6(a). LEG5 Joint Angle Motion for DATG.

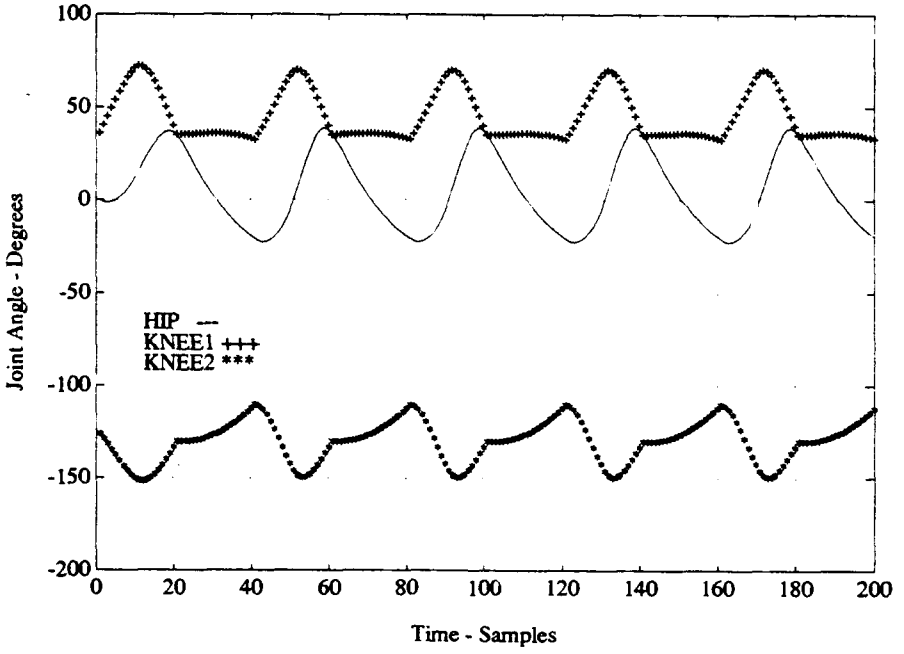


Figure 9-6(b). LEG5 Joint Angle Motion for CATG.

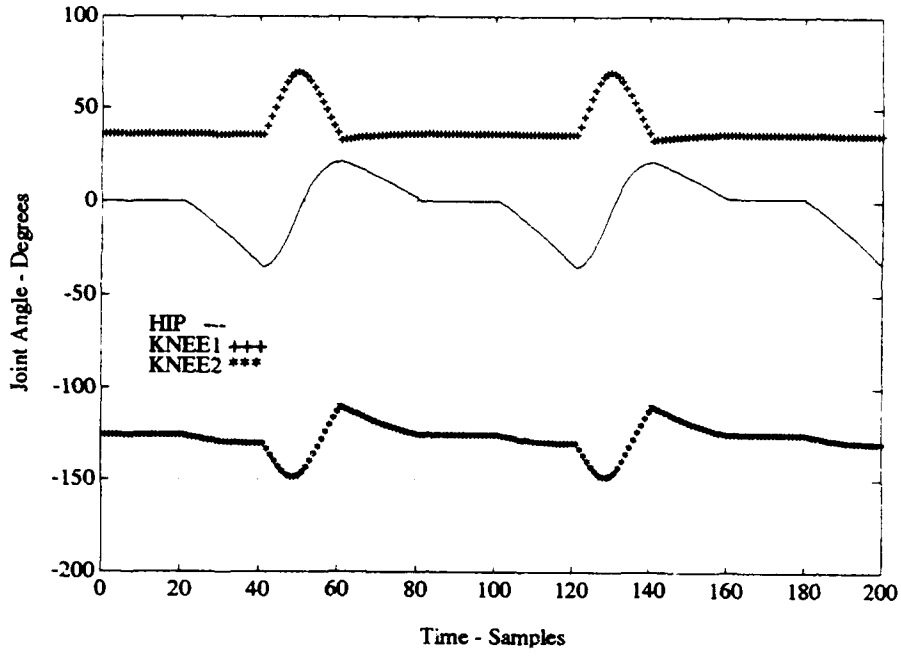


Figure 9-7(a). LEG6 Joint Angle Motion for DATG.

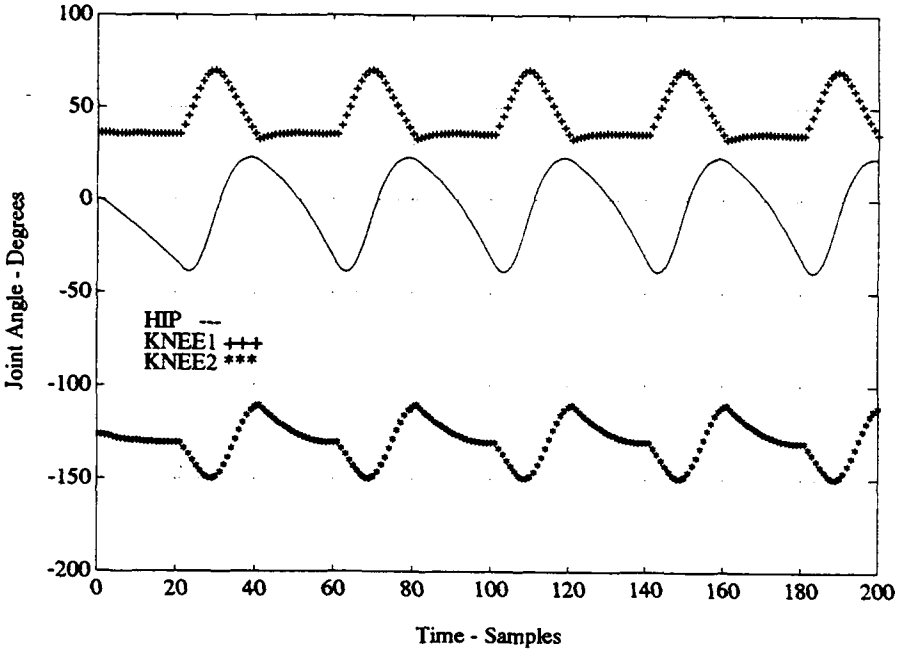


Figure 9-7(b). LEG6 Joint Angle Motion for CATG.

TRIPOD1. A comparison of joint angle motion for the two gait algorithms indicates:

- there are periods of no joint motion in the discrete gait,
- there are no such periods in the continuous gait, and
- in both gaits, there are discontinuities when the feet touch the ground.

The goals of this thesis, outlined in Chapter III, were to provide simulation of an improved alternating tripod gait that included:

- smooth leg motion,
- omnidirectional body movement,
- continuous body motion, and
- limited path following capability.

These goals have been met en route to the completion of this thesis.

B. RESEARCH CONTRIBUTIONS

This thesis is a direct outgrowth of a cooperative research effort between the Naval Postgraduate School and the Japanese Port and Harbor Research Institute to enhance the hardware and software capabilities of the *AquaRobot* underwater walking machine.

The PHRI originally developed *AquaRobot* to replace hard-hat divers constructing tsunami barriers in the hazardous underwater environment off Japan's coast. Although *AquaRobot* has demonstrated significant ability as the first hexapod underwater walking machine, it is still unable to perform its designed task more efficiently or less costly than human divers. *AquaRobot* speed and agility enhancements derived from this thesis will result in a more efficient and less costly machine. This translates directly into reduced risk of human injury and decreased construction costs.

Although walking machine research and construction is extensive, *AquaRobot* is the first walking robot with a practical use. As such, improvements to *AquaRobot* directly

result in concept fulfillment and favorably demonstrate the usefulness of robots, and walking robots in particular.

This opportunity to work and study with the PHRI lays the foundation for future cooperative U.S.-Japanese research activities in robotics. Even marginal headway towards improving U.S.-Japanese electronics and robotics research efforts will represent a significant cooperation milestone reached.

C. RESEARCH EXTENSIONS

1. Required Future Work

Towards the end of the thesis research, an opportunity to visit the PHRI in Japan presented itself [PHR93]. During the week-long visit, three significant dimensional changes to the *AquaRobot* were discovered that affect the workspace, stability, and maximum stride results in this thesis. The dimensional changes to *AquaRobot* were the result of ongoing research at the PHRI into the machine's mechanical attributes. None of the changes impact the derivation of the kinematics, inverse kinematics, stability, stride, or gait algorithms. However, the numerical values used in the calculations are slightly different from those culled from papers and blueprints. The three changes were:

- the footpad diameter increased from 25 centimeters to 45 centimeters,
- the footpad angle decreased from $\pm 45°$ to $\pm 22.5°$, and
- the lengths of LINK2 and LINK3 increased 1.8 and 1.0 centimeters, respectively.

The LINK dimension changes and the footpad angle change are reflected in Table 9-1. LINK0 and LINK1 were not changed. Figure 9-8 shows the reduced workspace resulting from the dimensional changes. The original workspace for LEG1 is shown with dashed lines to emphasize the change in working area.

The dimensional changes affect the workspace, stability, and maximum stride. First, the new workspace is reduced from its previous size. However, the workspace is used in the same manner as before. Next, stability is increased because the radius of the foot pad almost doubled. Because stability is measured at the center of the footpad, the additional radius actually imparts a greater stability safety margin. Finally, because of the footpad angle change, the maximum stride length is reduced from 178.21 centimeters to 147.57 centimeters. The total workspace area is reduced from 10367 to 5509 square centimeters, or approximately one-half its previous size.

TABLE 9-1. NEW LINK PARAMETERS FOR *AQUAROBOT*.

i	α_{i-1} (degrees)	a_{i-1} (cm)	d_i (cm)	θ_i (cm)	Range
-1	α_{WORLD}	a_{WORLD}	d_{WORLD}	θ_{WORLD}	
0	0	0	0	θ_0	$\theta_0 = 0°, 60°, 120°, 180°, 240°, 300°$
1	0	a_0	0	θ_1	$a_0 = 37.5$, $-60° \le \theta_1 \le 60°$
2	-90	a_1	0	θ_2	$a_1 = 20.0$, $-105.6° \le \theta_2 \le 74.4°$
3	0	a_2	0	θ_3	$a_2 = 51.8$, $-157.4° \le \theta_3 \le 22.6°$
4	0	a_3	0	θ_4	$a_3 = 101.0$, $-22.5° \le \theta_4 \le 22.5°$

a. Unconstrained Workspace

This thesis was developed using a constrained workspace. The workspace must eventually be unconstrained to determine which gaits, without sacrificing vehicle stability and energy efficiency and while maintaining continuous body motion, result in the fastest speeds for a hexapod underwater walking robot. This future work should primarily

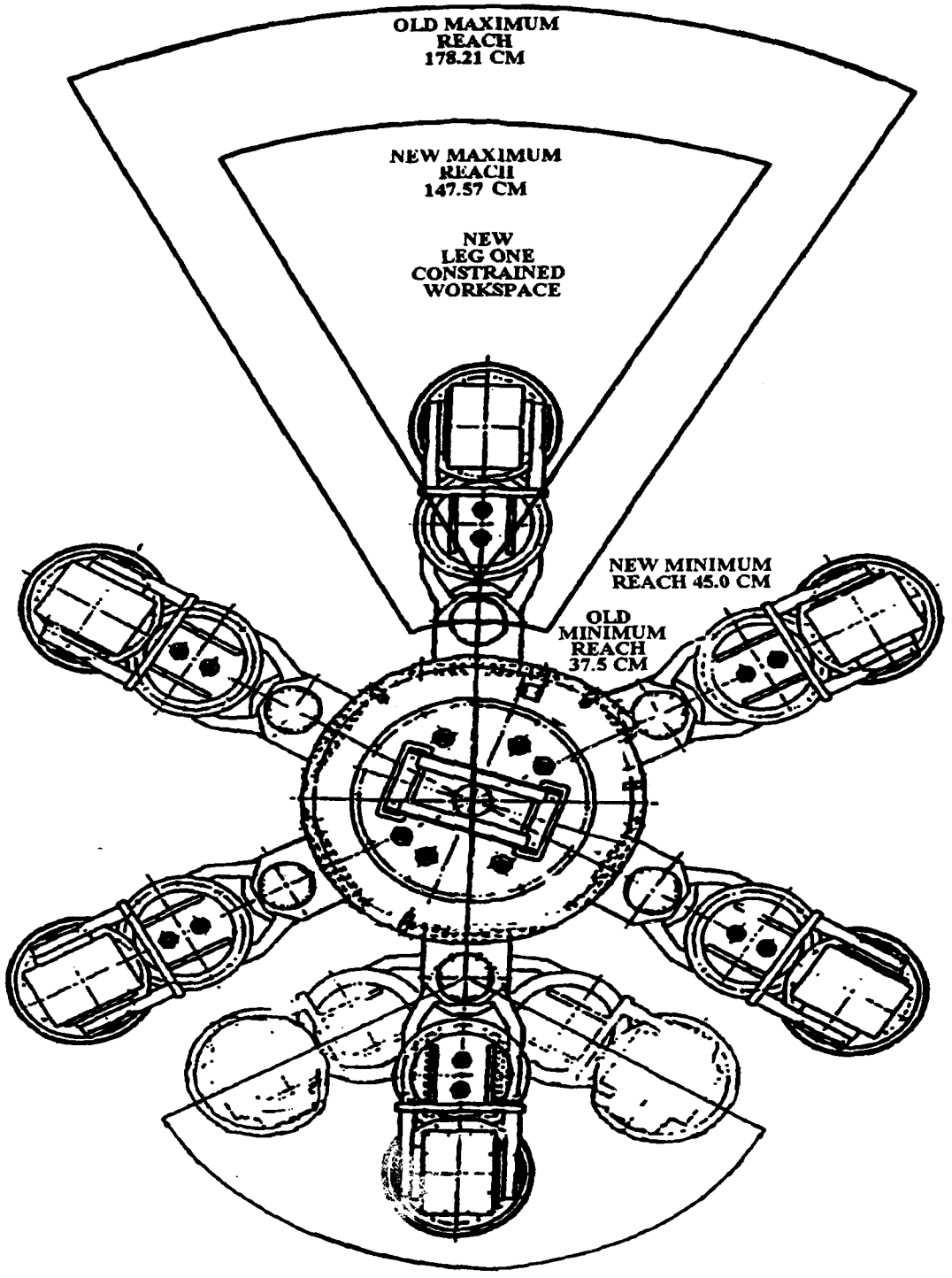


Figure 9-8. New Constrained Horizontal Workspace.

be directed towards investigating whether there is an analytical method for optimizing the leg cycling of a walking vehicle for an arbitrary given maneuvering specification while moving over rough terrain. Successful realization of this research would result in on-line optimization of foot placement.

b. Six DOF Body Movement

Statically stable alternating tripod gaits were planned and implemented in this thesis. To ease initial gait implementation, flat terrain, fixed body height, fixed body orientation, and straight line path constraints were placed on the vehicle. Constraints must gradually be removed to allow rough-terrain walking capability with an arbitrary orientation. Future work should include lifting the constraints on body movement: allow the body orientation and body height to change. If these efforts are completed, the *AquaRobot* simulation can then include navigation over irregular terrain.

c. Dynamic Effects

A study of the effects of the aqueous medium, ocean currents, tether, and camera boom loading on *AquaRobot* is advised. The greatest limiting effect on *AquaRobot's* speed may be the effects of the aqueous medium, including the effects of underwater currents, on the movement of the body and the legs. Environmental effects include the salt water medium itself and the approximately two knot subsurface ocean current typical off Japan's coast. Non-ideal characteristics include the effects of the tether and the main body camera boom on efficient walking.

2. Potential Future Work

The following paragraphs describe potential future work. The work is potential because research has shown that alternating tripod gaits are a simple way to achieve reasonable walking speed while maintaining static stability.

a. Other Stable Gaits

This thesis research concentrated on statically stable alternating tripod gaits. Additionally, only equilateral triangles (tripods) were considered. Considering isosceles, or other irregularly shaped, triangles may result in a more efficient use of stride and stability parameters towards increasing vehicle speed. Furthermore, because *AquaRobot's* six legs are located at 60 degree intervals, at least two bilateral symmetries are attainable. Symmetry is attained when a fore and aft pair of legs is defined or when any three legs are distinguished from the other three legs. One potentially important question is whether these alternative symmetries can be exploited to increase overall speed when arbitrary vehicle motion is desired.

b. Unstable (Free) Gaits

A potential future research topic includes determining whether there are optimality improvements when gaits are selected that result in momentary vehicle instability. The important question here is whether the viscous effects of the aqueous medium (especially the natural buoyancy of salt water) allows gaits which would be unstable in air to be used in the water.

APPENDIX A: *AQUAROBOT* OVERVIEW AND HISTORY

AquaRobot was conceived and designed at the Japanese Port and Harbor Research Institute (PHRI) in 1984. Its main purpose is to replace divers in carrying out underwater inspection work associated with various port construction projects. Specifically, the *AquaRobot* project is an outgrowth of a multi-billion dollar project to construct a tsunami barrier at the entrance to Japan's Kamaishi Bay. The primary design function of *AquaRobot* is to measure the flatness of underwater rubble mounds used as foundations to support concrete caissons that form the tsunami barrier. A secondary design function is to provide for observation of the underwat :r construction effort using a television camera.

Figure A-1 shows a representation of the expected underwater operation of *AquaRobot*. Navigational accuracy of *AquaRobot* is attained using a transponder system that allows position errors of less than ten centimeters at a distance of 300 meters [AKI89]. Every two steps, *AquaRobot* receives an updated navigational position from the transponder system [IWA90]. *AquaRobot* is tethered to a control platform, in this case a ship, with a 100 meter cable that includes optical fiber and metal wire links [IWA88b].

Three models of *AquaRobot* have been constructed to date. The first model, completed in 1985, was an experimental version developed for concept exploration and validation. It is used solely for overground testing. The experimental model weighs only 280 kilograms, yet can carry one person on its back. The second model, constructed in 1987, is the prototype version. The prototype is almost twice as large as the experimental model, mainly because of the increased leg length and the addition of the camera boom assembly, tether linkage, and lifting apparatus. The prototype is watertight to 50 meters depth, thereby causing additional bulk. The prototype weighs 700 kilograms. The third

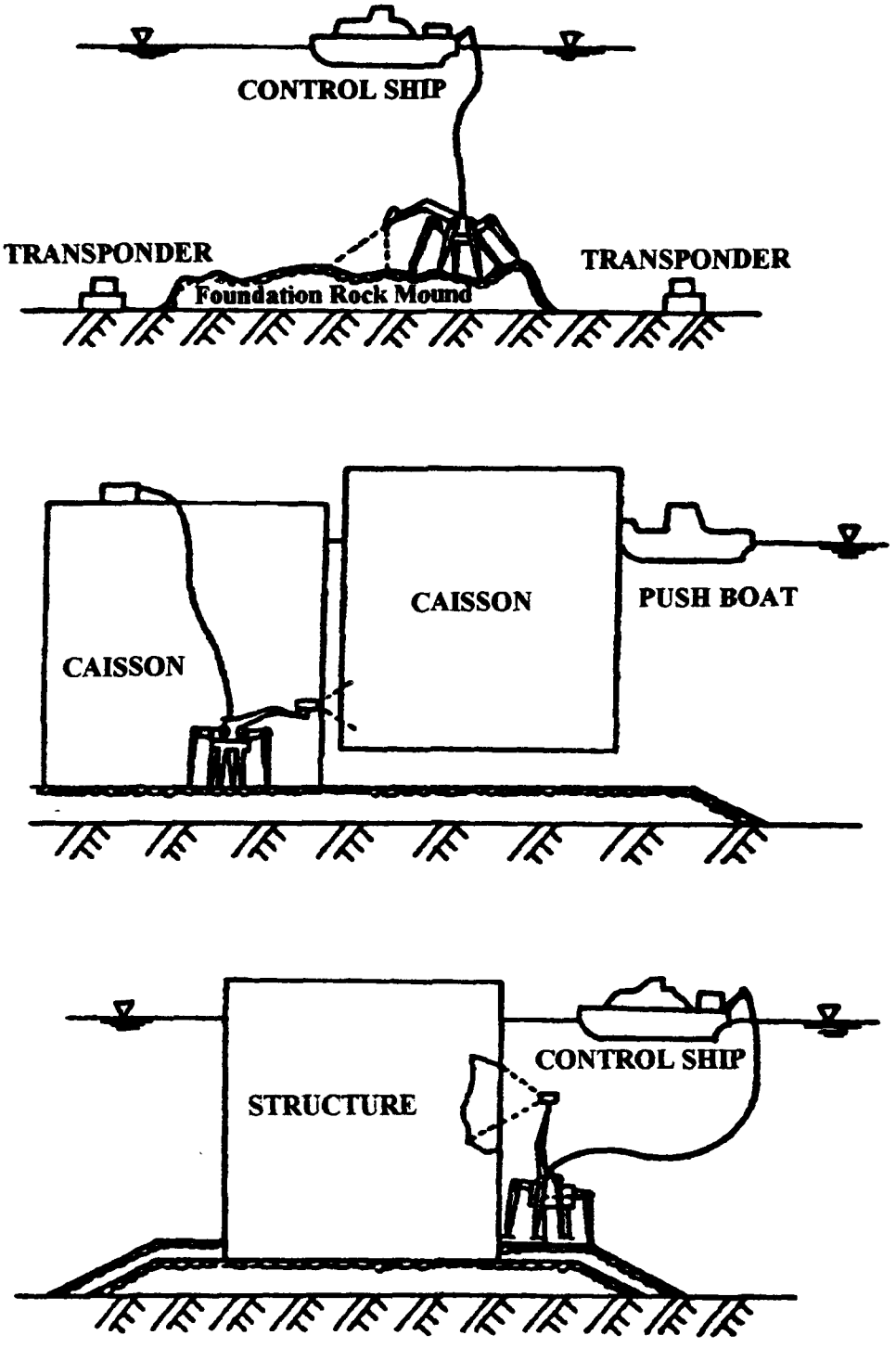


Figure A-1. Underwater Operation of *AquaRobot* (After [IWA88a]).

model, finished in 1989, is a light weight version of the prototype. Its improvements over the prototype include approximately half the weight and cable runs made inside leg articulations. [IWA90]

The body of *AquaRobot* is constructed of anti-corrosive aluminum and is hexagonal in shape. Six legs are attached at 60 degree increments about the body. Legs are numbered beginning with leg one and proceed in a clockwise direction through leg six. Each leg has three degrees of freedom and is constructed in a fashion similar to manufacturing manipulators. The three degrees of freedom are attained with three articulations, or joints per leg. The first joint rotates about a vertical axis and the next two joints rotate about horizontal axes. The joints are semi-direct drive with DC motor actuators and combination harmonic and bevel gears. Each of the 18 joint motors requires 70 Volt DC power, provided from the control platform via the tether cable. Each motor includes an encoder that produces 100 pulses per revolution. The motor control system is shown in Figure A-2. A motor driver for each motor counts the pulses from the controller, an 8086-based microcomputer, and from the motor encoder and drives the motor until the differential pulse count is zero. Using this method, the motors, and thereby, the joints, are rotated to the position the computer commands. [IWA88a]

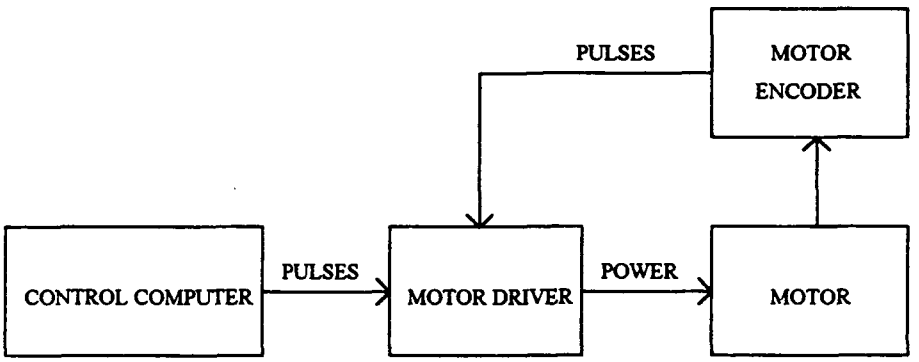


Figure A-2. Motor Control System (After [AKI89]).

Each of the six legs has an end-effector consisting of a 25 centimeter diameter, three centimeter thick, disk that acts like a foot. This foot operates from a passive ball-and-socket joint at the end of the leg. This passive joint constrains the motion between the long axis of the leg and the surface of the terrain to plus or minus 45 degrees.

An additional three-degree-of-freedom manipulator is attached to the top of the body and provides a platform for a TV camera and ultrasonic ranging device. Three lifting eyes are mounted equidistantly around the upper body and a tether interface box completes the top portion of the body.

AquaRobot uses four sensor types. A touch, or contact, sensor is installed near the junction of each leg and its foot. The contact sensors have a two millimeter plunger, but require only 0.5 millimeters of travel for activation [TAK93]. Two inclination sensors, or inclinometers, are installed in the body to determine the angle of inclination with respect to the terrain. A flux gate compass is also installed in the body to sense direction. Finally, a pressure sensor is included in the base of the body to accurately measure the water depth. [IWA88a]

AquaRobot's current control program is as shown in Figure A-3. The program includes walking and operating algorithms compiled and assembled in the BASIC programming language. There are essentially three sections of this program. First, there is the Walking Algorithm Program. The main purpose of this section of the program is to receive inputs from a human operator and convert those inputs into coordinate values for the robot's feet. Then the Robot Language section of the program is called. The Robot Language section includes fundamental command functions that perform operations such as changing the speed of motion of the leg, rotating the leg joints through a given angle, and determining whether a touch sensor has contacted the ground. The Robot Operating System Program then performs the necessary operations to actually move the robot.

Calculation of joint angles, conversion of angles to motor pulses, and providing the pulsed output to the motors is included. Finally, the Robot Operating System Program contains limited graphical display and simulation functions. [AKI89]

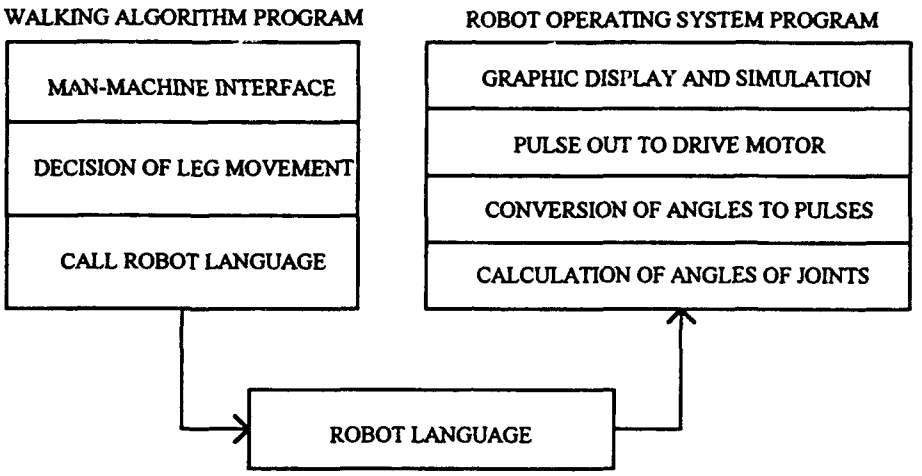


Figure A-3. *AquaRobot* Computer Control Program (After [AKI89]).

The prototype *AquaRobot* was actually tested in the field environment in February 1990. The test area chosen was the temporary storage area for completed caissons at the Izumi working area in the Kamaishi Port of the Iwate Prefecture. There, the sea bottom was covered with a rubble mound leveled with the same leveling machine used on the actual barrier site. The average depth was 24 meters. The navigational test is performed by having *AquaRobot* walk from one point to another along a previously measured path. Rectangularly shaped one-centimeter by two-centimeter steel plates are used to mark the waypoints along the path. Table A-1 shows the results of navigation and walking speed accuracy during the field test. Table A-1 shows a maximum error of +21 centimeters. This error includes the average navigational error of the transponder system, ± 10 centimeters. Errors may derive from aqueous influences (such as ocean currents), tether influences, or cumulative foot positioning offsets. From this table, it is obvious that walking speed

depends upon step height and step length. The maximum walking speed attained was 1.43 meters per minute. [IWA90]

TABLE A-1. NAVIGATION AND WALKING SPEED ACCURACY (AFTER [IWA90]).

Step Length (cm)	Step Height (cm)	Commanded Destination (cm)	Measured Destination (cm)	Error (cm)	Walking Speed (m/minute)
15	35	X 34821	X 34816	-5	0.61
		Y -7251	Y -7264	-13	
20	35	X 35029	X 35044	+15	0.67
		Y -8232	Y -8218	+14	
15	35	X 34821	X 34822	+1	0.67
		Y -7251	Y -7249	+2	
20	35	X 35029	X 35037	+8	0.76
		Y -8232	Y -8231	+1	
20	25	X 32810	X 32811	+1	1.27
		Y -7371	Y -7350	+21	
20	25	X 33000	X 32981	-19	1.43
		Y -7500	Y -7504	-4	

Figure A-4 shows the results of *AquaRobot* walking a planned path. The planned path included start and goal points and six waypoints. The actual walking distance was 95 meters. In this case, the average walking speed was 1.32 meters per minute. [IWA90]

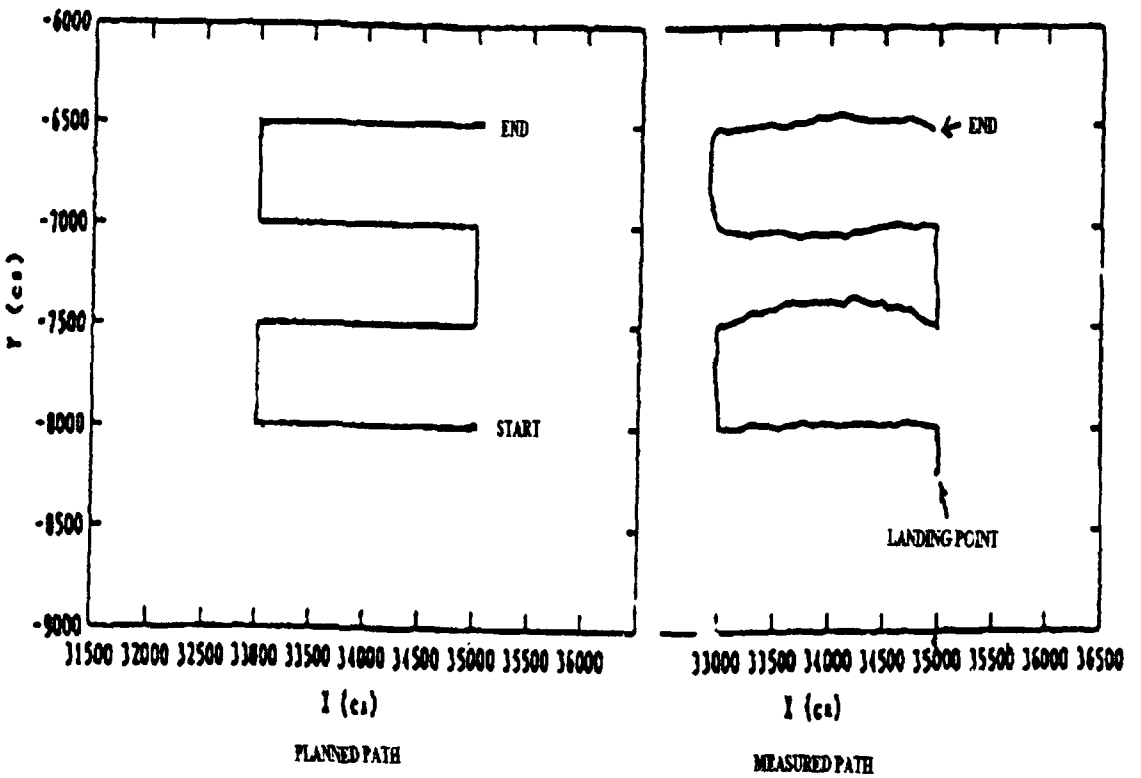


Figure A-4. Field Test of Planned and Measured Path (After [IWA90]).

APPENDIX B: MATLAB PROGRAM LISTINGS

This appendix contains the high level source code used in development and testing of the *AquaRobot* gait planning algorithms. The source code is in Matlab, a "C"-based development language. The Matlab code is not intended for subsequent use in either *AquaRobot* or the *AquaRobot* Simulator.

Matlab is a registered trademark of The Math Works, Incorporated. It does not produce stand-alone executable code, but is instead an interpretive program designed for numerical problem solving and is especially adept at matrix manipulation. The source code in this appendix requires Matlab for execution.

```
% ************************************************************************
% Title:   arcint.m
% Inputs:  (1,2,3,4,5,6,7)
% Outputs: [1,2,3]
% Purpose: This Matlab function determines the intersection of a ray
%          and an arc segment.
% ************************************************************************
%
function [intcpt, dist, flag] = arcint(foot, direc, cenbod, rad, limneg, limpos, ftnum)
%
% intcpt = (x,y) coordinates of intercept
% dist = distance between foot and intercepts
% flag = 1 if intercept found; flag = 0 if no intercept found
% foot = (x,y) foot components
% direc = direction of interest
% cenbod =  (x,y) coordinates of center of body
% rad = radius of circle
% limneg/limpos = endpoints of arc segment
% ftnum = foot 1 through 6
%
limit = 215; % maximum possible distance
flag = 0;
direc = direc*pi/180;
%
% Determine if the foot falls on the arc segment
footrad = sqrt(foot(1,1)^2 + foot(1,2)^2);
if (rad == footrad)
 flag = 1;
 intcpt = [foot(1,1), foot(1,2)];
 dist = limit;
end
%
if (flag == 0)
 perpd = ((cenbod(1,2) - foot(1,2))*cos(direc)) - ((cenbod(1,1) - foot(1,1))*sin(direc));
 if perpd <= rad
  length = sqrt(rad^2 - perpd^2);
  %
  % find (x,y) at intersection of perpd and length
  xperp = cenbod(1,1) + (perpd*sin(direc));
  yperp = cenbod(1,2) - (perpd*cos(direc));
  % test if foot is inside or outside radius
  radtest = sqrt((foot(1,2) - cenbod(1,2))^2 + (foot(1,1) - cenbod(1,1))^2);
  if (radtest > rad)
   % find (x,y) at intersection of ray and arc
   xint = xperp - (length*cos(direc));
   yint = yperp - (length*sin(direc));
   intcpt = [xint, yint];
  else
   % find (x,y) at intersection of ray and arc
   xint = xperp + (length*cos(direc));
   yint = yperp + (length*sin(direc));
   intcpt = [xint, yint];
  end
  %
  % test if intersection is in desired direction
  indir = foot(1,1)*cos(direc) + foot(1,2)*sin(direc);
  outdir = xint*cos(direc) + yint*sin(direc);
  if (indir <= outdir)
   dist = sqrt( (yint - foot(1,2))^2 + (xint - foot(1,1))^2 );
   %
   % test if intersection is within arc segment
   arcfoot = atan2( yint - cenbod(1,2), xint - cenbod(1,1) );
   narclim = atan2( limneg(1,2) - cenbod(1,2), limneg(1,1) - cenbod(1,1) );
   parclim = atan2( limpos(1,2) - cenbod(1,2), limpos(1,1) - cenbod(1,1) );
   if (ftnum == 1)
    if (arcfoot >= narclim) & (arcfoot <= parclim)
     flag = 1;
```

```
      end
     else
      if (direc > 0)
       if (narclim < 0)
        narclim = narclim + (2*pi);
       end
       if (parclim < 0)
        parclim = parclim + (2*pi);
       end
       if (arcfoot < 0)
        arcfoot = arcfoot + (2*pi);
       end
       if (arcfoot >= narclim) & (arcfoot <= parclim)
        flag = 1;
       end
      else  % (direc < 0)
       if (narclim > 0)
        narclim = narclim - (2*pi);
       end
       if (parclim > 0)
        parclim = parclim - (2*pi);
       end
       if (arcfoot > 0)
        arcfoot = arcfoot - (2*pi);
       end
       if (arcfoot >= narclim) & (arcfoot <= parclim)
        flag = 1;
       end
      end
     end
    end
   end
  end
end
if (flag ~= 1)
 intcpt = [];
 dist = limit;
 flag = 0;
end

% ************************************************************************
% Title:   arcint2.m
% Inputs:  (1,2,3,4,5,6)
% Outputs:  [1,2,3,4]
% Purpose:  This Matlab function determines the intersection of a directed
%         ray and an arc.
% ************************************************************************
%
function [xint,yint,dist,flag] = arcint2(rx,ry,rtheta,xc,yc,rad)
%
% rad = radius of circle(arc segment)
% xc/yc =  coordinates of center of circle
% rx/ry = ray components
% rtheta = direction of interest
% xint/yint = arc segment intercepts
% dist = distance between rx/ry and xint/yint
% flag = 1 if intercept found; flag = 0 if no intercept found
%
parclim=30;
narclim=30;
rtheta=(rtheta*pi/180);
%
d=(yc-ry)*cos(rtheta)-(xc-rx)*sin(rtheta);
if d<rad
 length=sqrt(rad^2-d^2);
elseif d==rad
 length=0;
 d=rad;
```

```
else
  xint=[];
  yint=[];
  dist=[];
  flag=0;
end
xperp=xc+(d*sin(rtheta));
yperp=yc-(d*cos(rtheta));
radtest=sqrt((ry-yc)^2+(rx-xc)^2);
if radtest>rad
  xint=xperp-(length*cos(rtheta));
  yint=yperp-(length*sin(rtheta));
elseif radtest<rad
  xint=xperp+(length*cos(rtheta));
  yint=yperp+(length*sin(rtheta));
% elseif radtest==rad
else
  flag=0;
end
%
% test if intersection is in desired direction
xpos=rad*sin(parclim*pi/180)+xc;
ypos=rad*cos(parclim*pi/180)+yc;
arcpos=atan2(ypos-yint,xpos-xint);
xneg=rad*sin(-narclim*pi/180)+xc;
yneg=rad*cos(-narclim*pi/180)+yc;
arcneg=atan2(yint-yneg,xint-xneg);
testpos=xpos*cos(arcpos)+ypos*sin(arcpos);
testneg=xneg*cos(arcneg)+yneg*sin(arcneg);
testintp=xint*cos(arcpos)+yint*sin(arcpos);
testintn=xint*cos(arcneg)+yint*sin(arcneg);
if (testintp<=testpos) & (testintn>=testneg)
  flag=1;
  dist=sqrt((yint-ry)^2+(xint-rx)^2);
else
  xint=[];
  yint=[];
  dist=[];
  flag=0;
end

% ************************************************************************
% Title:   b2wtrans.m
% Inputs:  (1,2,3,4,5,6,7,8,9)
% Outputs:  [1]
% Purpose:  This function transforms body coordinates to world coordinates.
% ************************************************************************
%
function [world] = b2wtrans(phi, theta, psi, xtrans, ytrans, ztrans, xb, yb, zb)
%
% phi = rotation of {B} about the Xw-axis;
% theta = rotation of {B} about the Yw-axis;
% psi = rotation of {B} about the Zw-axis;
% xtrans = X-axis translation of {B} with respect to {W};
% ytrans = Y-axis translation of {B} with respect to {W};
% ztrans = Z-axis translation of {B} with respect to {W};
% xb = known Xbody coordinate;
% yb = known Ybody coordinate;
% zb = known Zbody coordinate;
%
% construct the translation vector, the body vector, and the rotation matrix
trans = [xtrans, ytrans, ztrans]';
body = [xb,yb,zb,1]';
rotate = [cos(psi)*cos(theta) cos(psi)*sin(theta)*sin(phi)-sin(psi)*cos(phi) cos(psi)*sin(theta)*cos(phi)+sin(psi)*sin(phi);
        sin(psi)*cos(theta) sin(psi)*sin(theta)*sin(phi)+cos(psi)*cos(phi) sin(psi)*sin(theta)*cos(phi)-cos(psi)*sin(phi);
        -sin(theta) cos(theta)*sin(phi) cos(theta)*cos(phi)];
%
```

```
% build the body-to-world homogeneous matrix
BWT = [rotate trans;0 0 0 1];
%
% find the resultant 4X1 vector of equivalent world coordinates
Tworld = BWT*body;
%
% strip off the (x,y,z) coordinates
world = [Tworld(1,1), Tworld(2,1), Tworld(3,1)];

% ************************************************************************
% Title:   cycloid2.m
% Inputs:  radius a, stride d
% Outputs:
% Purpose: This Matlab calculates a segment of a cycloid and graphs it.
%
% ************************************************************************
%
clear
clc
clg
ain = 36.51; % input('Stride: ');
dtheta = input('NewFoot: ');
a = ain/(2*pi);
t = 0:a/200:2*pi;
x = a*(t - sin(t));
y = a*(1 - cos(t));
plot(x,y)
grid
title('Cycloid Foot Motion')
xlabel('Stride - cm')
ylabel('Height - cm')
hold on
theta = t(1,1)*pi/180;
dt = 2*dtheta*pi/180;
xnew = a*((theta+dt) - sin(theta+dt));
ynew = a*(1 - cos(theta+dt));
plot(xnew,ynew,'g+')
hold off

% ************************************************************************
% Title:   el3.m
% Inputs:
% Outputs:
% Purpose: This Matlab program calculates an elliptical foot path.
% ************************************************************************
%
% x^2/a^2 + y^2/b^2 =1, a>b
%
hold off
clc
clg
clear
pfx=input('Enter present foot-X: ');
pfy=0; %input('Enter present foot-Y: ');
dfx=input('Enter desired foot-X: ');
dfy=0; %input('Enter desired foot-Y: ');
a=(sqrt((dfy-pfy)^2+(dfx-pfx)^2))/2;
% set aspect
b=20;
x=-a:a/20:a;
if pfx<dfx
 xreal=pfx:(2*a)/(length(x)-1):dfx;
else
 xreal=pfx:-(2*a)/(length(x)-1):dfx;
end
elpse=[];
for p=1:length(x)
```

```
  y=b*sqrt(1-x(1,p)^2/a^2);
  elpse=[elpse;xreal(1,p) y];
end
axis([0 elpse(length(x),1)+50 elpse(1,2)-20 150])
plot(elpse(:,1),elpse(:,2),'g')
title('Elliptical Foot Path')
xlabel('X coordinates, (cm)')
ylabel('Y coordinates, (cm)')
hold on
plot(pfx,pfy,'oc8')
plot(dfx,dfy,'oc8')
grid

% ************************************************************************
% Title:   ellipse.m
% Inputs:  (1,2)
% Outputs:  [1,2,3]
% Purpose:  This Matlab function calculates an elliptical foot path using
%         the known foot position and the desired amount of foot
%         movement in the X direction.
% ************************************************************************
%
% x^2/a^2 + y^2/b^2 = 1; a > b
%
function [elpse, aspect, inc] = ellipse(pfx, dfx)
%
pfy = 0;
dfy = 0;
a = (sqrt((dfy-pfy)^2 + (dfx-pfx)^2))/2;
% set aspect
b = 20; % 20cm is maximum height (Z) or Link3 will overlap Link2
c = sqrt(abs(a^2 - b^2)); % absolute because b>a
inc = -a:a/20:a; % actual number of increments will probably be >500
if pfx < dfx
  xreal = pfx:(2*a)/(length(inc)-1):dfx;
else
  xreal = pfx:-(2*a)/(length(inc)-1):dfx;
end
elpse = [];
for p = 1:length(inc)
  y = b*sqrt(1 - inc(1,p)^2/a^2);
  elpse = [elpse;xreal(1,p) y];
end
aspect = c/a;

% ************************************************************************
% Title:   invkin.m
% Inputs:  (1,2,3)
% Outputs:  [1,2,3,4]
% Purpose:  This Matlab function provides the inverse kinematics of the
%         AquaRobot for a given foot location (px,py,pz) and a chosen
%         leg number (1-6).
% ************************************************************************
%
function [theta0d,theta1d,theta2d,theta3d]=invkin(leg,px,py,pz)
%
% theta0'd'=degrees; without 'd'=radians
%
a0=37.5;
a1=20;
a2=50;
a3=100;
%
if leg==1
  theta0=0*pi/180;
elseif leg==2
  theta0=-60*pi/180;
```

```
elseif leg==3
 theta0=-120*pi/180;
elseif leg==4
 theta0=-180*pi/180;
elseif leg==5
 theta0=-240*pi/180;
elseif leg==6
 theta0=-300*pi/180;
else
 theta0=0*pi/180;
end
theta0d=theta0*180/pi;
%
b1=sqrt((px-a0*cos(theta0))^2+(py-a0*sin(theta0))^2);
c1=(px^2+py^2-a0^2-b1^2)/(2*a0*b1);
theta1d=atan2(sqrt(1-c1^2),c1)*180/pi;
%
b2=b1-a1;
b3=sqrt(b2^2+pz^2);
c2=(a2^2+b3^2-a3^2)/(2*a2*b3);
theta2m=atan2(-sqrt(1-c2^2),c2)-atan2(pz,b2);
theta2=atan2(sqrt(1-c2^2),c2)-atan2(pz,b2);
if (theta2<=106.6*pi/180)&(theta2>=-73.4*pi/180)
     theta2d=theta2*180/pi;
else
     theta2=theta2m;
     theta2d=theta2*180/pi;
end;
%
c3=(b3^2-a2^2-a3^2)/(2*a2*a3);
theta3m=atan2(-sqrt(1-c3^2),c3);
theta3=atan2(sqrt(1-c3^2),c3);
%if (theta3<=156.4*pi/180)&(theta3>=-23.6*pi/180)
%      theta3=theta3m;
%      theta3d=theta3*180/pi;
%else
%      theta3d=theta3*180/pi;
%end;
if theta3>=0*180/pi
 theta3d=theta3*180/pi;
else
 theta3=theta3m;
 theta3d=theta3*180/pi;
end;

% ************************************************************************
% Title:   jcoord.m
% Inputs:
% Outputs:
% Purpose:  This Matlab program checks the kinematics of the AquaRobot.
%         Joints Zero, One, Two, and Three are used.
% ************************************************************************
%
!del c
diary c
angle0=input('Enter Leg angle (0,60,120,180,240,300): ');
angle1=input('Enter Hip angle                : ');
angle2=input('Enter Knee1 angle              : ');
angle3=input('Enter Knee2 angle              : ');
diary off
%
angle0=angle0*pi/180;
angle1=angle1*pi/180;
angle2=angle2*pi/180;
angle3=angle3*pi/180;
%
a0=37.5;
```

```
a1=20;
a2=50;
a3=100;
%
TB0=[cos(angle0) -sin(angle0) 0 0;
   sin(angle0) cos(angle0) 0 0;
   0 0 1 0;
   0 0 0 1];
%
T01=[cos(angle1) -sin(angle1) 0 a0;
   sin(angle1) cos(angle1) 0 0;
   0 0 1 0;
   0 0 0 1];
%
diary c
HIP=TB0*T01
pause
diary off
%
T12=[cos(angle2) -sin(angle2) 0 a1;
   0 0 1 0;
   -sin(angle2) -cos(angle2) 0 0;
   0 0 0 1];
%
diary c
KNEE1=TB0*T01*T12
pause
diary off
%
T23=[cos(angle3) -sin(angle3) 0 a2;
   sin(angle3) cos(angle3) 0 0;
   0 0 1 0;
   0 0 0 1];
%
diary c
KNEE2=TB0*T01*T12*T23
pause
diary off
%
T34=[1 0 0 a3; -
   0 1 0 0;
   0 0 1 0;
   0 0 0 1];
%
diary c
FOOT=TB0*T01*T12*T23*T34
diary off

% ************************************************************************
% Title:   jkin0123.m
% Inputs:
% Outputs:
% Purpose:  This Matlab program checks the kinematics of the AquaRobot.
%         Joints Zero, One, Two, and Three are used.
% ************************************************************************
%
!del c
diary c
angle0=input('Enter Leg angle (0,60,120,180,240,300):  ');
angle1=input('Enter Hip angle                  :  ');
angle2=input('Enter Knee1 angle                :  ');
angle3=input('Enter Knee2 angle                :  ');
diary off
%
angle0=angle0*pi/180;
angle1=angle1*pi/180;
angle2=angle2*pi/180;
```

```
angle3=angle3*pi/180;
%
a0=37.5;
a1=20;
a2=50;
%
TB0=[cos(angle0) -sin(angle0) 0 0;
   sin(angle0) cos(angle0) 0 0;
   0 0 1 0;
   0 0 0 1];
%
T01=[cos(angle1) -sin(angle1) 0 a0;
   sin(angle1) cos(angle1) 0 0;
   0 0 1 0;
   0 0 0 1];
%
diary c
HIP=TB0*T01
pause
diary off
%
T12=[cos(angle2) -sin(angle2) 0 a1;
   0 0 1 0;
   -sin(angle2) -cos(angle2) 0 0;
   0 0 0 1];
%
diary c
KNEE1=TB0*T01*T12
pause
diary off
%
T23=[cos(angle3) -sin(angle3) 0 a2;
   sin(angle3) cos(angle3) 0 0;
   0 0 1 0;
   0 0 0 1];
%
diary c
KNEE2=TB0*T01*T12*T23
pause
diary off

% ************************************************************************
% Title:   jkin123.m
% Inputs:
% Outputs:
% Purpose:  This Matlab program checks the kinematics of the AquaRobot.
%         Joints One, Two, and Three are used.
% ************************************************************************
%
!del c
diary c
angle1=input('Enter Hip angle                : ');
angle2=input('Enter Knee1 angle              : ');
angle3=input('Enter Knee2 angle              : ');
diary off
%
angle1=angle1*pi/180;
angle2=angle2*pi/180;
angle3=angle3*pi/180;
%
a0=37.5;
a1=20;
a2=50;
a3=100;
%
T01=[cos(angle1) -sin(angle1) 0 a0;
   sin(angle1) cos(angle1) 0 0;
```

```
   0 0 1 0;
   0 0 0 1];

T12=[cos(angle2) -sin(angle2) 0 a1;
   0 0 1 0;
   -sin(angle2) -cos(angle2) 0 0;
   0 0 0 1];

diary c
KNEE1=T01*T12
pause
diary off

T23=[cos(angle3) -sin(angle3) 0 a2;
   sin(angle3) cos(angle3) 0 0;
   0 0 1 0;
   0 0 0 1];

diary c
KNEE2=T01*T12*T23
pause
diary off

% ************************************************************************
% Title:   kin2.m
% Inputs:  (1,2,3,4)
% Outputs:  [1,2,3,4]
% Purpose:  This Matlab function provides the kinematics of the AquaRobot
%        for a given set of joint angles. Uses new LINK lengths.
% ************************************************************************
%
function [Hip,Knee1,Knee2,Foot] = kin2(theta0,theta1,theta2,theta3)
%
angle0=theta0*pi/180;
angle1=theta1*pi/180;
angle2=theta2*pi/180;
angle3=theta3*pi/180;
%
a0=37.5;
a1=21.5;
a2=52;
a3=101;
%
TB0=[cos(angle0) -sin(angle0) 0 0;
   sin(angle0) cos(angle0) 0 0;
   0 0 1 0;
   0 0 0 1];
%
T01=[cos(angle1) -sin(angle1) 0 a0;
   sin(angle1) cos(angle1) 0 0;
   0 0 1 0;
   0 0 0 1];
%
H=TB0*T01;
%
T12=[cos(angle2) -sin(angle2) 0 a1;
   0 0 1 0;
   -sin(angle2) -cos(angle2) 0 0;
   0 0 0 1];
%
K1=TB0*T01*T12;
%
T23=[cos(angle3) -sin(angle3) 0 a2;
   sin(angle3) cos(angle3) 0 0;
   0 0 1 0;
   0 0 0 1];
%
```

```
K2=TB0*T01*T12*T23;
%
T34=[1 0 0 a3;
   0 1 0 0;
   0 0 1 0;
   0 0 0 1];
%
FT=TB0*T01*T12*T23*T34;
%
Hip=[H(1,4) H(2,4) H(3,4)];
Knee1=[K1(1,4) K1(2,4) K1(3,4)];
Knee2=[K2(1,4) K2(2,4) K2(3,4)];
Foot=[FT(1,4) FT(2,4) FT(3,4)];

% ************************************************************************
% Title:   maxd.m
% Inputs:  (1,2,3,4,5,6,7,8,9)
% Outputs: [1]
% Purpose: This Matlab function finds the maximum stride possible for
%          AquaRobot, given an arbitrary direction as an input.
% ************************************************************************
%
function [maxdist]=maxd(foot1,foot2,foot3,foot4,foot5,foot6,direc,cenbod,tri)
%
% maxdist = maximum foot movement in the direction requested
% foot1:foot6 = (x,y) coordinates of feet
% direc = direction of interest
% cenbod = (x,y) coordinates of the center of the body
% tri = tripod number 0 (legs 1,3,5) or number 2 (legs 2,4,6)
%
% ***** MUST CONVERT (X,Y,Z)WORLD COORDINATES TO BODY COORDINATES HERE *****
%
% initializations here
%
inrad = 37.5;      % inner radius
outrad = 178.2107;  % outer radius
%
seg1pin = [36.8686, 6.8524];  % inside endpoint of positive segment #1
seg1pout = [160.2049, 78.0606]; % outside endpoint of positive segment #1
seg1nin = [36.8686, -6.8524];  % inside endpoint of negative segment #1
seg1nout = [160.2049, -78.0606]; % outside endpoint of negative segment #1
%
seg2pin = [12.5, 35.3553];  % inside endpoint of positive segment #2
seg2pout = [12.5, 177.7718]; % outside endpoint of positive segment #2
seg2nin = [24.3686, 28.5030];  % inside endpoint of negative segment #2
seg2nout = [147.7049, 99.7112]; % outside endpoint of negative segment #2
%
seg3pin = [-24.3686, 28.5030];  % inside endpoint of positive segment #3
seg3pout = [-147.7049, 99.7112]; % outside endpoint of positive segment #3
seg3nin = [-12.5, 35.3553];  % inside endpoint of negative segment #3
seg3nout = [-12.5, 177.7718]; % outside endpoint of negative segment #3
%
seg4pin = [-36.8686, -6.8524];  % inside endpoint of positive segment #4
seg4pout = [-160.2049, -78.0606]; % outside endpoint of positive segment #4
seg4nin = [-36.8686, 6.8524];  % inside endpoint of negative segment #4
seg4nout = [-160.2049, 78.0606]; % outside endpoint of negative segment #4
%
seg5pin = [-12.5, -35.3553];  % inside endpoint of positive segment #5
seg5pout = [-12.5, -177.7718]; % outside endpoint of positive segment #5
seg5nin = [-24.3686, -28.5030];  % inside endpoint of negative segment #5
seg5nout = [-147.7049, -99.7112]; % outside endpoint of negative segment #5
%
seg6pin = [24.3686, -28.5030];  % inside endpoint of positive segment #6
seg6pout = [147.7049, -99.7112]; % outside endpoint of positive segment #6
seg6nin = [12.5, -35.3553];  % inside endpoint of negative segment #6
seg6nout = [12.5, -177.7718]; % outside endpoint of negative segment #6
%
```

```
flag = 0;
while (tri == 0)
 % test foot1 in leg 1 workspace
 ft = 1;
 [intcpt, dist1, flag]=arcint(foot1, direc, cenbod, outrad, seg1nout, seg1pout, ft);
 if (flag == 1)
  dista = dist1;
 end
 [intcpt, dist1, flag]=arcint(foot1, direc, cenbod, inrad, seg1nin, seg1pin, ft);
 if (flag == 1)
  dista = dist1;
 end
 [intcpt, dist1, flag]=segint(foot1, direc, seg1pin, seg1pout);
 if (flag == 1)
  dista = dist1;
 end
 [intcpt, dist1, flag]=segint(foot1, direc, seg1nin, seg1nout);
 if (flag == 1)
  dista = dist1;
 end
 %
 % test foot3 in leg 3 workspace
 ft = 3;
 [intcpt, dist3, flag]=arcint(foot3, direc, cenbod, outrad, seg3nout, seg3pout, ft);
 if (flag == 1)
  distb = dist3;
 end
 [intcpt, dist3, flag]=arcint(foot3, direc, cenbod, inrad, seg3nin, seg3pin, ft);
 if (flag == 1)
  distb = dist3;
 end
 [intcpt, dist3, flag]=segint(foot3, direc, seg3pin, seg3pout);
 if (flag == 1)
  distb = dist3;
 end
 [intcpt, dist3, flag]=segint(foot3, direc, seg3nin, seg3nout);
 if (flag == 1)
  distb = dist3;
 end
 %
 % test foot5 in leg 5 workspace
 ft = 5;
 [intcpt, dist5, flag]=arcint(foot5, direc, cenbod, outrad, seg5nout, seg5pout, ft);
 if (flag == 1)
  distc = dist5;
 end
 [intcpt, dist5, flag]=arcint(foot5, direc, cenbod, inrad, seg5nin, seg5pin, ft);
 if (flag == 1)
  distc = dist5;
 end
 [intcpt, dist5, flag]=segint(foot5, direc, seg5pin, seg5pout);
 if (flag == 1)
  distc = dist5;
 end
 [intcpt, dist5, flag]=segint(foot5, direc, seg5nin, seg5nout);
 if (flag == 1)
  distc = dist5;
 end
 %
 maxdist = min( min(dista, distb), distc);
 %
 break;
end  % this ends tripod 0 testing
%
% ================================================================
while (tri == 2)
 % test foot2 in leg 2 workspace
```

```
ft = 2;
[intcpt, dist2, flag]=arcint(foot2, direc, cenbod, outrad, seg2nout, seg2pout, ft);
if (flag == 1)
  dista = dist2;
end
[intcpt, dist2, flag]=arcint(foot2, direc, cenbod, inrad, seg2nin, seg2pin, ft);
if (flag == 1)
  dista = dist2;
end
[intcpt, dist2, flag]=segint(foot2, direc, seg2pin, seg2pout);
if (flag == 1)
  dista = dist2;
end
[intcpt, dist2, flag]=segint(foot2, direc, seg2nin, seg2nout);
if (flag == 1)
  dista = dist2;
end
%
% test foot4 in leg 4 workspace
ft = 4;
[intcpt, dist4, flag]=arcint(foot4, direc, cenbod, outrad, seg4nout, seg4pout, ft);
if (flag == 1)
  distb = dist4;
end
[intcpt, dist4, flag]=arcint(foot4, direc, cenbod, inrad, seg4nin, seg4pin, ft);
if (flag == 1)
  distb = dist4;
end
[intcpt, dist4, flag]=segint(foot4, direc, seg4pin, seg4pout);
if (flag == 1)
  distb = dist4;
end
[intcpt, dist4, flag]=segint(foot4, direc, seg4nin, seg4nout);
if (flag == 1)
  distb = dist4;
end
%
% test foot6 in leg 6 workspace
ft = 6;
[intcpt, dist6, flag]=arcint(foot6, direc, cenbod, outrad, seg6nout, seg6pout, ft);
if (flag == 1)
  distc = dist6;
end
[intcpt, dist6, flag]=arcint(foot6, direc, cenbod, inrad, seg6nin, seg6pin, ft);
if (flag == 1)
  distc = dist6;
end
[intcpt, dist6, flag]=segint(foot6, direc, seg6pin, seg6pout);
if (flag == 1)
  distc = dist6;
end
[intcpt, dist6, flag]=segint(foot6, direc, seg6nin, seg6nout);
if (flag == 1)
  distc = dist6;
end
%
maxdist = min( min(dista, distb), distc);
%
break;
end  % this ends tripod 1 testing

% ************************************************************************
% Title:   maxd25.m
% Inputs:  (1,2,3,4,5,6,7,8,9)
% Outputs:  [1]
% Purpose:  This Matlab function finds the maximum stride possible for
%         AquaRobot, given an arbitrary direction as an input.
```

```
%        This function is for the 25 cm footpad.
% ********************************************************************
%
function [maxdist]=maxd25(foot1,foot2,foot3,foot4,foot5,foot6,direc,cenbod,tri)
%
% maxdist = maximum foot movement in the direction requested
% foot1:foot6 = (x,y) coordinates of feet
% direc = direction of interest
% cenbod = (x,y) coordinates of the center of the body
% tri = tripod number 0 (legs 1,3,5) or number 2 (legs 2,4,6)
%
% ***** MUST CONVERT (X,Y,Z)WORLD COORDINATES TO BODY COORDINATES HERE *****
%
% initializations here
%
inrad = 37.5;      % inner radius
outrad = 178.2107;  % outer radius
%
seg1pin = [36.8686, -6.8524];  % inside endpoint of positive segment #1
seg1pout = [160.2049, -78.0606]; % outside endpoint of positive segment #1
seg1nin = [36.8686, 6.8524];  % inside endpoint of negative segment #1
seg1nout = [160.2049, 78.0606]; % outside endpoint of negative segment #1
%
seg6pin = [12.5, 35.3553];  % inside endpoint of positive segment #2
seg6pout = [12.5, 177.7718]; % outside endpoint of positive segment #2
seg6nin = [24.3686, 28.5030];  % inside endpoint of negative segment #2
seg6nout = [147.7049, 99.7112]; % outside endpoint of negative segment #2
%
seg5pin = [-24.3686, 28.5030];  % inside endpoint of positive segment #3
seg5pout = [-147.7049, 99.7112]; % outside endpoint of positive segment #3
seg5nin = [-12.5, 35.3553];  % inside endpoint of negative segment #3
seg5nout = [-12.5, 177.7718]; % outside endpoint of negative segment #3
%
seg4pin = [-36.8686, 6.8524];  % inside endpoint of positive segment #4
seg4pout = [-160.2049, 78.0606]; % outside endpoint of positive segment #4
seg4nin = [-36.8686, -6.8524];  % inside endpoint of negative segment #4
seg4nout = [-160.2049, -78.0606]; % outside endpoint of negative segment #4
%
seg3pin = [-12.5, -35.3553];  % inside endpoint of positive segment #5
seg3pout = [-12.5, -177.7718]; % outside endpoint of positive segment #5
seg3nin = [-24.3686, -28.5030];  % inside endpoint of negative segment #5
seg3nout = [-147.7049, -99.7112]; % outside endpoint of negative segment #5
%
seg2pin = [24.3686, -28.5030];  % inside endpoint of positive segment #6
seg2pout = [147.7049, -99.7112]; % outside endpoint of positive segment #6
seg2nin = [12.5, -35.3553];  % inside endpoint of negative segment #6
seg2nout = [12.5, -177.7718]; % outside endpoint of negative segment #6
%
flag = 0;
while (tri == 0)
 % test foot1 in leg 1 workspace
 ft = 1;
 [intcpt, dist1, flag]=arcint(foot1, direc, cenbod, outrad, seg1nout, seg1pout, ft);
 if (flag == 1)
   dista = dist1;
 end
 [intcpt, dist1, flag]=arcint(foot1, direc, cenbod, inrad, seg1nin, seg1pin, ft);
 if (flag == 1)
   dista = dist1;
 end
 [intcpt, dist1, flag]=segint(foot1, direc, seg1pin, seg1pout);
 if (flag == 1)
   dista = dist1;
 end
 [intcpt, dist1, flag]=segint(foot1, direc, seg1nin, seg1nout);
 if (flag == 1)
   dista = dist1;
```

```
 end
 %
 % test foot3 in leg 3 workspace
 ft = 3;
 [intcpt, dist3, flag]=arcint(foot3, direc, cenbod, outrad, seg3nout, seg3pout, ft);
 if (flag == 1)
  distb = dist3;
 end
 [intcpt, dist3, flag]=arcint(foot3, direc, cenbod, inrad, seg3nin, seg3pin, ft);
 if (flag == 1)
  distb = dist3;
 end
 [intcpt, dist3, flag]=segint(foot3, direc, seg3pin, seg3pout);
 if (flag == 1)
  distb = dist3;
 end
 [intcpt, dist3, flag]=segint(foot3, direc, seg3nin, seg3nout);
 if (flag == 1)
  distb = dist3;
 end
 %
 % test foot5 in leg 5 workspace
 ft = 5;
 [intcpt, dist5, flag]=arcint(foot5, direc, cenbod, outrad, seg5nout, seg5pout, ft);
 if (flag == 1)
  distc = dist5;
 end
 [intcpt, dist5, flag]=arcint(foot5, direc, cenbod, inrad, seg5nin, seg5pin, ft);
 if (flag == 1)
  distc = dist5;
 end
 [intcpt, dist5, flag]=segint(foot5, direc, seg5pin, seg5pout);
 if (flag == 1)
  distc = dist5;
 end
 [intcpt, dist5, flag]=segint(foot5, direc, seg5nin, seg5nout);
 if (flag == 1)
  distc = dist5;
 end
 %
 maxdist = min( min(dista, distb), distc);
 %
 break;
end  % this ends tripod 0 testing
%
% ================================================================
while (tri == 2)
 % test foot2 in leg 2 workspace
 ft = 2;
 [intcpt, dist2, flag]=arcint(foot2, direc, cenbod, outrad, seg2nout, seg2pout, ft);
 if (flag == 1)
  dista = dist2;
 end
 [intcpt, dist2, flag]=arcint(foot2, direc, cenbod, inrad, seg2nin, seg2pin, ft);
 if (flag == 1)
  dista = dist2;
 end
 [intcpt, dist2, flag]=segint(foot2, direc, seg2pin, seg2pout);
 if (flag == 1)
  dista = dist2;
 end
 [intcpt, dist2, flag]=segint(foot2, direc, seg2nin, seg2nout);
 if (flag == 1)
  dista = dist2;
 end
 %
 % test foot4 in leg 4 workspace
```

```
 ft = 4;
 [intcpt, dist4, flag]=arcint(foot4, direc, cenbod, outrad, seg4nout, seg4pout, ft);
 if (flag == 1)
  distb = dist4;
 end
 [intcpt, dist4, flag]=arcint(foot4, direc, cenbod, inrad, seg4nin, seg4pin, ft);
 if (flag == 1)
  distb = dist4;
 end
 [intcpt, dist4, flag]=segint(foot4, direc, seg4pin, seg4pout);
 if (flag == 1)
  distb = dist4;
 end
 [intcpt, dist4, flag]=segint(foot4, direc, seg4nin, seg4nout);
 if (flag == 1)
  distb = dist4;
 end
 %
 % test foot6 in leg 6 workspace
 ft = 6;
 [intcpt, dist6, flag]=arcint(foot6, direc, cenbod, outrad, seg6nout, seg6pout, ft);
 if (flag == 1)
  distc = dist6;
 end
 [intcpt, dist6, flag]=arcint(foot6, direc, cenbod, inrad, seg6nin, seg6pin, ft);
 if (flag == 1)
  distc = dist6;
 end
 [intcpt, dist6, flag]=segint(foot6, direc, seg6pin, seg6pout);
 if (flag == 1)
  distc = dist6;
 end
 [intcpt, dist6, flag]=segint(foot6, direc, seg6nin, seg6nout);
 if (flag == 1)
  distc = dist6;
 end
 %
 maxdist = min( min(dista, distb), distc);
 %
 break;
end  % this ends tripod 1 testing

% ************************************************************************
% Title:   maxd45.m
% Inputs:  (1,2,3,4,5,6,7,8,9)
% Outputs:  [1]
% Purpose:  This Matlab function finds the maximum stride possible for
%         AquaRobot, given an arbitrary direction as an input.
%         This function is for the 45 cm footpad.
% ************************************************************************
%
function [maxdist]=maxd45(foot1,foot2,foot3,foot4,foot5,foot6,direc,cenbod,tri)
%
% maxdist = maximum foot movement in the direction requested
% foot1:foot6 = (x,y) coordinates of feet
% direc = direction of interest
% cenbod = (x,y) coordinates of the center of the body
% tri = tripod number 0 (legs 1,3,5) or number 2 (legs 2,4,6)
%
% ***** MUST CONVERT (X,Y,Z)WORLD COORDINATES TO BODY COORDINATES HERE *****
%
% initializations here
%
inrad = 45.0;      % inner radius
outrad = 149.27;  % outer radius
%
seg1pin = [45.0, 0.0];  % inside endpoint of positive segment #1
```

```
seg1pout = [139.0446, 54.2967]; % outside endpoint of positive segment #1
seg1nin = [45.0, 0.0];  % inside endpoint of negative segment #1
seg1nout = [139.0446, -54.2967]; % outside endpoint of negative segment #1
%
seg2pin = [22.5, 38.9711];  % inside endpoint of positive segment #2
seg2pout = [22.5, 147.5645]; % outside endpoint of positive segment #2
seg2nin = [22.5, 38.9711];  % inside endpoint of negative segment #2
seg2nout = [116.5446, 93.2678]; % outside endpoint of negative segment #2
%
seg3pin = [-22.5, 38.9711];  % inside endpoint of positive segment #3
seg3pout = [-116.5446, 93.2678]; % outside endpoint of positive segment #3
seg3nin = [-22.5, 38.9711];  % inside endpoint of negative segment #3
seg3nout = [-22.5, 147.5645]; % outside endpoint of negative segment #3
%
seg4pin = [-45.0, 0.0];  % inside endpoint of positive segment #4
seg4pout = [-139.0446, -54.2967]; % outside endpoint of positive segment #4
seg4nin = [-45.0, 0.0];  % inside endpoint of negative segment #4
seg4nout = [-139.0446, 54.2967]; % outside endpoint of negative segment #4
%
seg5pin = [-22.5, -38.9711];  % inside endpoint of positive segment #5
seg5pout = [-22.5, -147.5645]; % outside endpoint of positive segment #5
seg5nin = [-22.5, -38.9711];  % inside endpoint of negative segment #5
seg5nout = [-116.5446, -93.2678]; % outside endpoint of negative segment #5
%
seg6pin = [22.5, -38.9711];  % inside endpoint of positive segment #6
seg6pout = [116.5446, -93.2678]; % outside endpoint of positive segment #6
seg6nin = [22.5, -38.9711];  % inside endpoint of negative segment #6
seg6nout = [22.5, -147.5645]; % outside endpoint of negative segment #6
%
flag = 0;
while (tri == 0)
 % test foot1 in leg 1 workspace
 ft = 1;
 [intcpt, dist1, flag]=arcint(foot1, direc, cenbod, outrad, seg1nout, seg1pout, ft);
 if (flag == 1)
  dista = dist1;
 end
 [intcpt, dist1, flag]=segint(foot1, direc, seg1pin, seg1pout);
 if (flag == 1)
  dista = dist1;
 end
 [intcpt, dist1, flag]=segint(foot1, direc, seg1nin, seg1nout);
 if (flag == 1)
  dista = dist1;
 end
 %
 % test foot3 in leg 3 workspace
 ft = 3;
 [intcpt, dist3, flag]=arcint(foot3, direc, cenbod, outrad, seg3nout, seg3pout, ft);
 if (flag == 1)
  distb = dist3;
 end
 [intcpt, dist3, flag]=segint(foot3, direc, seg3pin, seg3pout);
 if (flag == 1)
  distb = dist3;
 end
 [intcpt, dist3, flag]=segint(foot3, direc, seg3nin, seg3nout);
 if (flag == 1)
  distb = dist3;
 end
 %
 % test foot5 in leg 5 workspace
 ft = 5;
 [intcpt, dist5, flag]=arcint(foot5, direc, cenbod, outrad, seg5nout, seg5pout, ft);
 if (flag == 1)
  distc = dist5;
 end
```

```
 [intcpt, dist5, flag]=segint(foot5, direc, seg5pin, seg5pout);
 if (flag == 1)
  distc = dist5;
 end
 [intcpt, dist5, flag]=segint(foot5, direc, seg5nin, seg5nout);
 if (flag == 1)
  distc = dist5;
 end
 %
 maxdist = min( min(dista, distb), distc);
 %
 break;
end  % this ends tripod 0 testing
%
% ================================================================
while (tri == 2)
 % test foot2 in leg 2 workspace
 ft = 2;
 [intcpt, dist2, flag]=arcint(foot2, direc, cenbod, outrad, seg2nout, seg2pout, ft);
 if (flag == 1)
  dista = dist2;
 end
 [intcpt, dist2, flag]=segint(foot2, direc, seg2pin, seg2pout);
 if (flag == 1)
  dista = dist2;
 end
 [intcpt, dist2, flag]=segint(foot2, direc, seg2nin, seg2nout);
 if (flag == 1)
  dista = dist2;
 end
 %
 % test foot4 in leg 4 workspace
 ft = 4;
 [intcpt, dist4, flag]=arcint(foot4, direc, cenbod, outrad, seg4nout, seg4pout, ft);
 if (flag == 1)
  distb = dist4;
 end
 [intcpt, dist4, flag]=segint(foot4, direc, seg4pin, seg4pout);
 if (flag == 1)
  distb = dist4;
 end
 [intcpt, dist4, flag]=segint(foot4, direc, seg4nin, seg4nout);
 if (flag == 1)
  distb = dist4;
 end
 %
 % test foot6 in leg 6 workspace
 ft = 6;
 [intcpt, dist6, flag]=arcint(foot6, direc, cenbod, outrad, seg6nout, seg6pout, ft);
 if (flag == 1)
  distc = dist6;
 end
 [intcpt, dist6, flag]=segint(foot6, direc, seg6pin, seg6pout);
 if (flag == 1)
  distc = dist6;
 end
 [intcpt, dist6, flag]=segint(foot6, direc, seg6nin, seg6nout);
 if (flag == 1)
  distc = dist6;
 end
 %
 maxdist = min( min(dista, distb), distc);
 %
 break;
end  % this ends tripod 1 testing

% ************************************************************************
```

```
% Title:   md.m
% Inputs:  (1,2,3,4,5,6,7,8,9)
% Outputs: [1]
% Purpose: This Matlab function finds the maximum stride possible for
%          AquaRobot, given an arbitrary direction as an input.
%          This function is for the 25 cm footpad.
% ********************************************************************
%
function [maxdist]=md(foot1,foot2,foot3,foot4,foot5,foot6,direc,cenbod,tri)
%
% maxdist = maximum foot movement in the direction requested
% foot1:foot6 = (x,y) coordinates of feet
% direc = direction of interest
% cenbod = (x,y) coordinates of the center of the body
% tri = tripod number 0 (legs 1,3,5) or number 2 (legs 2,4,6)
%
inrad = 37.5;      % inner radius of workspace
outrad = 178.2107;  % outer radius of workspace
%
seg1nin = [36.8686, -6.8524];  % inside endpoint of positive segment #1
seg1nout = [160.2049, -78.0606]; % outside endpoint of positive segment #1
seg1pin = [36.8686, 6.8524];  % inside endpoint of negative segment #1
seg1pout = [160.2049, 78.0606]; % outside endpoint of negative segment #1
%
seg6nin = [12.5, -35.3553];  % inside endpoint of positive segment #
seg6nout = [12.5, -177.7718]; % outside endpoint of positive segment #
seg6pin = [24.3686, -28.5030];  % inside endpoint of negative segment #
seg6pout = [147.7049, -99.7112]; % outside endpoint of negative segment #
%
seg5nin = [-24.3686, -28.5030];  % inside endpoint of positive segment #
seg5nout = [-147.7049, -99.7112]; % outside endpoint of positive segment #
seg5pin = [-12.5, -35.3553];  % inside endpoint of negative segment #
seg5pout = [-12.5, -177.7718]; % outside endpoint of negative segment #
%
seg4nin = [-36.8686, 6.8524];  % inside endpoint of positive segment #
seg4nout = [-160.2049, 78.0606]; % outside endpoint of positive segment #
seg4pin = [-36.8686, -6.8524];  % inside endpoint of negative segment #
seg4pout = [-160.2049, -78.0606]; % outside endpoint of negative segment #
%
seg3nin = [-12.5,-35.3553];  % inside endpoint of positive segment #
seg3nout = [-12.5, 177.7718]; % outside endpoint of positive segment #
seg3pin = [-24.3686, 28.5030];  % inside endpoint of negative segment #
seg3pout = [-147.7049, 99.7112]; % outside endpoint of negative segment #
%
seg2nin = [24.3686, 28.5030];  % inside endpoint of positive segment #
seg2nout = [147.7049, 99.7112]; % outside endpoint of positive segment #
seg2pin = [12.5, 35.3553];  % inside endpoint of negative segment #
seg2pout = [12.5, 177.7718]; % outside endpoint of negative segment #
%
flag = 0;
while (tri == 0)
 % test foot1 in leg 1 workspace
 ft = 1;
 [intcpt, dist1, flag]=arcint(foot1, direc, cenbod, outrad, seg1nout, seg1pout, ft);
 if (flag == 1)
   dista = dist1;
 end
 [intcpt, dist1, flag]=arcint(foot1, direc, cenbod, inrad, seg1nin, seg1pin, ft);
 if (flag == 1)
   dista = dist1;
 end
 rotate = 0;
 [intcpt, dist1, flag]=segint(foot1, direc, seg1pin, seg1pout, rotate); % CCW segment
 if (flag == 1)
   dista = dist1;
 end
 rotate = 1;
```

```
[intcpt, dist1, flag]=segint(foot1, direc, seg1nin, seg1nout, rotate); % CW segment
if (flag == 1)
  dista = dist1;
end
%
% test foot3 in leg 3 workspace
ft = 3;
[intcpt, dist3, flag]=arcint(foot3, direc, cenbod, outrad, seg3nout, seg3pout, ft);
if (flag == 1)
  distb = dist3;
end
[intcpt, dist3, flag]=arcint(foot3, direc, cenbod, inrad, seg3nin, seg3pin, ft);
if (flag == 1)
  distb = dist3;
end
rotate = 0;
[intcpt, dist3, flag]=segint(foot3, direc, seg3pin, seg3pout, rotate); % CCW segment
if (flag == 1)
  distb = dist3;
end
rotate = 1;
[intcpt, dist3, flag]=segint(foot3, direc, seg3nin, seg3nout, rotate); % CW segment
if (flag == 1)
  distb = dist3;
end
%
% test foot5 in leg 5 workspace
ft = 5;
[intcpt, dist5, flag]=arcint(foot5, direc, cenbod, outrad, seg5nout, seg5pout, ft);
if (flag == 1)
  distc = dist5;
end
[intcpt, dist5, flag]=arcint(foot5, direc, cenbod, inrad, seg5nin, seg5pin, ft);
if (flag == 1)
  distc = dist5;
end
rotate = 0;
[intcpt, dist5, flag]=segint(foot5, direc, seg5pin, seg5pout, rotate); % CCW segment
if (flag == 1)
  distc = dist5;
end
rotate = 1;
[intcpt, dist5, flag]=segint(foot5, direc, seg5nin, seg5nout, rotate); % CW segment
if (flag == 1)
  distc = dist5;
end
%
% fprintf('mstuff,' L1 = %5.1f,dista);
% fprintf('mstuff,' L3 = %5.1f,distb);
% fprintf('mstuff,' L5 = %5.1f,distc);
maxdist = min( min(dista, distb), distc);
%
break;
end  % this ends tripod 0 testing
%
% ================================================================
while (tri == 2)
 % test foot2 in leg 2 workspace
 ft = 2;
 [intcpt, dist2, flag]=arcint(foot2, direc, cenbod, outrad, seg2nout, seg2pout, ft);
 if (flag == 1)
   dista = dist2;
 end
 [intcpt, dist2, flag]=arcint(foot2, direc, cenbod, inrad, seg2nin, seg2pin, ft);
 if (flag == 1)
   dista = dist2;
 end
```

```
rotate = 0;
[intcpt, dist2, flag]=segint(foot2, direc, seg2pin, seg2pout, rotate); % CCW segment
if (flag == 1)
  dista = dist2;
end
rotate = 1;
[intcpt, dist2, flag]=segint(foot2, direc, seg2nin, seg2nout, rotate); % CW segment
if (flag == 1)
  dista = dist2;
end
%
% test foot4 in leg 4 workspace
ft = 4;
[intcpt, dist4, flag]=arcint(foot4, direc, cenbod, outrad, seg4nout, seg4pout, ft);
if (flag == 1)
  distb = dist4;
end
[intcpt, dist4, flag]=arcint(foot4, direc, cenbod, inrad, seg4nin, seg4pin, ft);
if (flag == 1)
  distb = dist4;
end
rotate = 0;
[intcpt, dist4, flag]=segint(foot4, direc, seg4pin, seg4pout, rotate); % CCW segment
if (flag == 1)
  distb = dist4;
end
rotate = 1;
[intcpt, dist4, flag]=segint(foot4, direc, seg4nin, seg4nout, rotate); % CW segment
if (flag == 1)
  distb = dist4;
end
%
% test foot6 in leg 6 workspace
ft = 6;
[intcpt, dist6, flag]=arcint(foot6, direc, cenbod, outrad, seg6nout, seg6pout, ft);
if (flag == 1)
  distc = dist6;
end
[intcpt, dist6, flag]=arcint(foot6, direc, cenbod, inrad, seg6nin, seg6pin, ft);
if (flag == 1)
  distc = dist6;
end
rotate = 0;
[intcpt, dist6, flag]=segint(foot6, direc, seg6pin, seg6pout, rotate); % CCW segment
if (flag == 1)
  distc = dist6;
end
rotate = 1;
[intcpt, dist6, flag]=segint(foot6, direc, seg6nin, seg6nout, rotate); % CW segment
if (flag == 1)
  distc = dist6;
end
%
% fprintf('mstuff,' L2 = %5.1f,dista);
% fprintf('mstuff,' L4 = %5.1f,distb);
% fprintf('mstuff,' L6 = %5.1f,distc);
 maxdist = min( min(dista, distb), distc);
 %
 break;
end % this ends tripod 1 testing

% ************************************************************************
% Title:   mk.m
% Purpose: Tests maxdist routines at +/- ? degree intervals.
% ************************************************************************
%
function mk()
```

```
%
ft1=[98.02,0];
ft2=[12.5,85]; %[49.01,84.8878];
ft3=[-49.01,84.8878];
ft4=[-97,0]; %[-98.02,0];
ft5=[-49.01,-84.8878];
ft6=[12.5,-85]; %[49.01,-84.8878];
cenbod=[0,0];
%
!del mstuff
%
tri = 0;
fprintf('mstuff','        Tripod 0 - POSITIVE\n');
fprintf('mstuff','=============================================================\n\n');
for direc = 0:90:360
 fprintf('mstuff','d = %4.0f',direc);
 [m] = md(ft1,ft2,ft3,ft4,ft5,ft6,direc,cenbod,tri);
 m
 fprintf('mstuff',' max = %5.1f\n',m);
end
%
fprintf('mstuff','\n        Tripod 0 - NEGATIVE\n');
fprintf('mstuff','=============================================================\n\n');
for direc = -0:-90:-360
 fprintf('mstuff','d = %4.0f',direc);
 [m] = md(ft1,ft2,ft3,ft4,ft5,ft6,direc,cenbod,tri);
 m
 fprintf('mstuff',' max = %5.1f\n',m);
end
%
tri = 2;
%
fprintf('mstuff','\n\n\n        Tripod 1 - POSITIVE\n');
fprintf('mstuff','=============================================================\n\n');
for direc = 0:1:360
 fprintf('mstuff','d = %4.0f',direc);
 [m] = md(ft1,ft2,ft3,ft4,ft5,ft6,direc,cenbod,tri);
 m
 fprintf('mstuff',' max = %5.1f\n',m);
end
%
fprintf('mstuff','\n        Tripod 1 - NEGATIVE\n');
fprintf('mstuff','=============================================================\n\n');
for direc = -0:-1:-360
 fprintf('mstuff','d = %4.0f',direc);
 [m] = md(ft1,ft2,ft3,ft4,ft5,ft6,direc,cenbod,tri);
 m
 fprintf('mstuff',' max = %5.1f\n',m);
end
```

This Matlab file contains the | -angles
parameters used in the maxd.m, |
segint.m, and arcint.m functions. ——+—— +X
NOTE: This is NOT an executable |
program. | +angles
+Y

params.m; 27APR93; 25cm foot only

These parameters were derived using the arcint2.m function called
as shown using Leg ONE inputs as an example:

[x,y,d,f] = arcint2(25,0,30,0,0,37.5).

Leg ONE (x, y): workspace angles defined from (25, 0)
radius inner arc 37.5 cm:
positive inner segment (36.8686, 6.8524); 30 degrees
negative inner segment (36.8686, -6.8524); 330 degree

radius outer arc 178.2107 cm:
positive outer segment (160.2049, 78.0606); 30 degrees
negative outer segment (160.2049, -78.0606); 330 degrees

Leg TWO (x, y): workspace angles defined from (12.5, 21.6506)
radius inner arc 37.5 cm:
positive inner segment (12.5, 35.3553); 90 degrees
negative inner segment (24.3686, 28.5030); 30 degrees
radius outer arc 178.2107 cm:
positive outer segment (12.5, 177.7718); 90 degrees
negative outer segment (147.7049, 99.7112); 30 degrees

Leg THREE (x, y): workspace angles defined from (-12.5, 21.6506)
radius inner arc 37.5 cm:
positive inner segment (-24.3686, 28.5030); 150 degrees
negative inner segment (-12.5, 35.3553); 90 degrees
radius outer arc 178.2107 cm:
positive outer segment (-147.7049, 99.7112); 150 degrees
negative outer segment (-12.5, 177.7718); 90 degrees

Leg FOUR (x, y): workspace angles defined from (-25, 0)
radius inner arc 37.5 cm:
positive inner segment (-36.8686, -6.8524); 210 degrees
negative inner segment (-36.8686, 6.8524); 150 degrees
radius outer arc 178.2107 cm:
positive outer segment (-160.2049, -78.0606); 210 degrees
negative outer segment (-160.2049, 78.0606); 150 degrees

Leg FIVE (x, y): workspace angles defined from (-12.5, -21.6506)
radius inner arc 37.5 cm:
positive inner segment (-12.5, -35.3553); 270 degrees
negative inner segment (-24.3686, -28.5030); 210 degrees
radius outer arc 178.2107 cm:
positive outer segment (-12.5, -177.7718); 270 degrees
negative outer segment (-147.7049, -99.7112); 210 degrees

Leg SIX (x, y): workspace angles defined from (12.5, -21.6506)
radius inner arc 37.5 cm:
positive inner segment (24.3686, -28.5030); 330 degrees
negative inner segment (12.5, -35.3553); 270 degrees
radius outer arc 178.2107 cm:
positive outer segment (147.7049, -99.7112); 330 degrees
negative outer segment (12.5, -177.7718); 270 degrees

This Matlab file contains the parameters used in the maxd3.m, segint3.m, and arcint3.m functions.
NOTE: This is NOT an executable program.

params3.m; 26APR93; 45cm foot only

These parameters were derived using the arcint2.m function called as shown using Leg ONE inputs as an example:

[x,y,d,f] = arcint2(25,0,30,0,0,37.5).

Leg ONE (x, y): workspace angles defined from (45, 0)
radius outer arc 149.27 cm:
positive outer segment (139.0446, 54.2967); 30 degrees
negative outer segment (139.0446, -54.2967); 330 degrees

Leg TWO (x, y): workspace angles defined from (22.5, 38.9711)
radius outer arc 149.27 cm:
positive outer segment (22.5, 147.5645); 90 degrees
negative outer segment (116.5446, 93.2678); 30 degrees

```
Leg THREE (x, y):  workspace angles defined from (-22.5, 38.9711)
 radius outer arc 149.27 cm:
  positive outer segment (-116.5446, 93.2678);  150 degrees
  negative outer segment (-22.5, 147.5645);  90 degrees

Leg FOUR (x, y):  workspace angles defined from (-45, 0)
 radius outer arc 149.27 cm:
  positive outer segment (-139.0446, -54.2967);  210 degrees
  negative outer segment (-139.0446, 54.2967);  150 degrees

Leg FIVE (x, y):  workspace angles defined from (-22.5, -38.9711)
 radius outer arc 149.2707 cm:
  positive outer segment (-22.5, -147.5645);  270 degrees
  negative outer segment (-116.5446, -93.2678);  210 degrees

Leg SIX (x, y):  workspace angles defined from (22.5, -38.9711)
 radius outer arc 149.27 cm:
  positive outer segment (116.5446, -93.2678);  330 degrees
  negative outer segment (22.5, -147.5645);  270 degrees

% ************************************************************************
% Title:   pltfoot.m
% Inputs:
% Outputs:  plt.met plot
% Purpose:  This Matlab program plots the progression of KNEE2 and the
%         FOOT of AquaRobot given a known X foot coordinate and the
%         desired leg stride.
% ************************************************************************
%
clear
clg
clc
!del plt.met
hold off
yglobal=-70.7107;
%
px = input('Present foot X: ');
%
% NOTE: use next line for direct entry of next x-coordinate, OR
% dx = input('Desired foot X: ');
% NOTE: use next two lines to enter stepsize from known foot point
xin = input('Desired stride: ');
dx = px + xin;
%
[elpse, e, elx] = ellipse(px, dx);
%
hip = [];
knee1 = [];
knee2 = [];
foot = [];
for n = 1:length(elpse)
  elpse(n,2) = elpse(n,2) + yglobal;
  [t0,t1,t2,t3] = invkin(1,elpse(n,1),0,elpse(n,2));
  [h,k1,k2,ft]=kin(t0,t1,t2,t3);
  knee2=[knee2;k2(1,1) k2(1,3)];
  foot=[foot;ft(1,1) ft(1,3)];
end
hip=[h(1,1) h(1,3)];
knee1=[k1(1,1) k1(1,3)];
axis([-10 180 -100 60])
plot(hip(1,1),hip(1,2),'+c2')
hold on
title('Elliptical Foot Motion')
xlabel('AquaRobot X-coordinates')
ylabel('AquaRobot Z-coordinates')
%
plot(knee1(1,1),knee1(1,2),'+c5')
```

```
%
plot(knee2(1,1),knee2(1,2),'*c8')
for p=2:n-1
 plot(knee2(p,1),knee2(p,2),'+g')
end
plot(knee2(n,1),knee2(n,2),'xc8')
%
plot(foot(1,1),foot(1,2),'*c6')
for p=2:n-1
 plot(foot(p,1),foot(p,2),'oc6')
end
plot(foot(n,1),foot(n,2),'xc6')
%
text(0.3,0.58,'Hip','sc')
text(0.4,0.58,'K1','sc')
text(0.2,0.18,'* - start','sc')
text(0.2,0.15,'x - goal','sc')
grid
meta plt

% ************************************************************************
% Title:   reset.m
% Inputs:
% Outputs:  kcR.met plot; ks data file
% Purpose:  This Matlab program plots the RESET posture for AquaRobot.
% ************************************************************************
%
!del ks
!del kcR.met
clear
hold off
%
angle01=0*pi/180;  % leg1
angle02=60*pi/180;  % leg2
angle03=120*pi/180; % leg3
angle04=180*pi/180; % leg4
angle05=240*pi/180; % leg5
angle06=300*pi/180; % leg6
%
a0=37.5;
a1=20;
a2=50;
a3=100;
%
angle1=0;
%
angle2F=-66.4*pi/180;
angle2B=66.4*pi/180;
%
angle3F=156.4*pi/180;
angle3B=-156.4*pi/180;
%
T00=[cos(angle01) -sin(angle01) 0 0;sin(angle01) cos(angle01) 0 0;0 0 1 0;0 0 0 1];
T01=[cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP=T00*T01;
T12=[cos(angle2F) -sin(angle2F) 0 a1;0 0 1 0;-sin(angle2F) -cos(angle2F) 0 0;0 0 0 1];
KNEE1=T00*T01*T12;
T23=[cos(angle3F) -sin(angle3F) 0 a2;sin(angle3F) cos(angle3F) 0 0;0 0 1 0;0 0 0 1];
KNEE2=T00*T01*T12*T23;
T34=[1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT1=T00*T01*T12*T23*T34;
diary off
%
T00=[cos(angle02) -sin(angle02) 0 0;sin(angle02) cos(angle02) 0 0;0 0 1 0;0 0 0 1];
T01=[cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP=T00*T01;
```

```
T12=[cos(angle2F) -sin(angle2F) 0 a1;0 0 1 0;-sin(angle2F) -cos(angle2F) 0 0;0 0 0 1];
KNEE1=T00*T01*T12;
T23=[cos(angle3F) -sin(angle3F) 0 a2;sin(angle3F) cos(angle3F) 0 0;0 0 1 0;0 0 0 1];
KNEE2=T00*T01*T12*T23;
T34=[1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT2=T00*T01*T12*T23*T34;
diary off
%
T00=[cos(angle03) -sin(angle03) 0 0;sin(angle03) cos(angle03) 0 0;0 0 1 0;0 0 0 1];
T01=[cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP=T00*T01;
T12=[cos(angle2B) -sin(angle2B) 0 a1;0 0 1 0;-sin(angle2B) -cos(angle2B) 0 0;0 0 0 1];
KNEE1=T00*T01*T12;
T23=[cos(angle3B) -sin(angle3B) 0 a2;sin(angle3B) cos(angle3B) 0 0;0 0 1 0;0 0 0 1];
KNEE2=T00*T01*T12*T23;
T34=[1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT3=T00*T01*T12*T23*T34;
diary off
%
T00=[cos(angle04) -sin(angle04) 0 0;sin(angle04) cos(angle04) 0 0;0 0 1 0;0 0 0 1];
T01=[cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP=T00*T01;
T12=[cos(angle2B) -sin(angle2B) 0 a1;0 0 1 0;-sin(angle2B) -cos(angle2B) 0 0;0 0 0 1];
KNEE1=T00*T01*T12;
T23=[cos(angle3B) -sin(angle3B) 0 a2;sin(angle3B) cos(angle3B) 0 0;0 0 1 0;0 0 0 1];
KNEE2=T00*T01*T12*T23;
T34=[1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT4=T00*T01*T12*T23*T34;
diary off
%
T00=[cos(angle05) -sin(angle05) 0 0;sin(angle05) cos(angle05) 0 0;0 0 1 0;0 0 0 1];
T01=[cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP=T00*T01;
T12=[cos(angle2B) -sin(angle2B) 0 a1;0 0 1 0;-sin(angle2B) -cos(angle2B) 0 0;0 0 0 1];
KNEE1=T00*T01*T12;
T23=[cos(angle3B) -sin(angle3B) 0 a2;sin(angle3B) cos(angle3B) 0 0;0 0 1 0;0 0 0 1];
KNEE2=T00*T01*T12*T23;
T34=[1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT5=T00*T01*T12*T23*T34;
diary off
%
T00=[cos(angle06) -sin(angle06) 0 0;sin(angle06) cos(angle06) 0 0;0 0 1 0;0 0 0 1];
T01=[cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP=T00*T01;
T12=[cos(angle2F) -sin(angle2F) 0 a1;0 0 1 0;-sin(angle2F) -cos(angle2F) 0 0;0 0 0 1];
KNEE1=T00*T01*T12;
T23=[cos(angle3F) -sin(angle3F) 0 a2;sin(angle3F) cos(angle3F) 0 0;0 0 1 0;0 0 0 1];
KNEE2=T00*T01*T12*T23;
T34=[1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT6=T00*T01*T12*T23*T34;
diary off
%
x=[FOOT1(1,4) FOOT2(1,4) FOOT3(1,4) FOOT4(1,4) FOOT5(1,4) FOOT6(1,4)];
y=[FOOT1(2,4) FOOT2(2,4) FOOT3(2,4) FOOT4(2,4) FOOT5(2,4) FOOT6(2,4)];
axis([-200 200 -200 200])
plot(x,y,'o')
hold
x=0;
y=0;
plot(x,y,'+g')
xlabel('X Foot Coordinates (cm)')
ylabel('Y Foot Coordinates (cm)')
```

```
title('AquaRobot RESET Posture')
grid
gtext('FOOT1')
gtext('FOOT2')
gtext('Body Center')
meta kcR

% This MATLAB program finds the intersection of the
% desired direction of travel and the workspace.
%
% Written by Chuck Schue, 29DEC92
% rl.m
%
% Notes: 1. Aquarobot positive angle/rotation is CCW.
%        2. Matlab positive angle/rotation is CW.
%        3. Leg One
%
clear
clc
clg
Flag=0;
%=================================================================
% Declarations
Rxn=157.6844;
Ryn=76.6054;
Rxp=157.6844;
Ryp=-76.6054;
%=================================================================
% Define line (pos_leg_workspace_limit)
thetapin=-30;
thetap=thetapin*pi/180;
xlinep=25+(12.5*cos(thetap));
ylinep=12.5*sin(thetap);
%=================================================================
% Define line (neg_leg_workspace_limit)
thetanin=30;
thetan=thetanin*pi/180;
xlinen=25+(12.5*cos(thetan));
ylinen=12.5*sin(thetan);
%=================================================================
% Define arc (max_workspace_limit) with radius 178.2107
%
cmax=[];
r=178;
x=[160:0.5:178]; % actually 157.6844 to 178.2107
for m=1:length(x)
   y=sqrt(r^2-x(:,m)^2);
   cmax=[cmax;x(:,m) y];
end
x=[178:-0.5:160]; % actually 157.6844 to 178.2107
for n=1:length(x)
   y=-sqrt(r^2-x(:,n)^2);
   cmax=[cmax;x(:,n) y];
end
%=================================================================
% Define arc (min_workspace_limit) with radius 37.5
%
cmin=[];
r=37.5;
x=[36.9755:0.1:37.5];
for m=1:length(x)
   y=sqrt(r^2-x(:,m)^2);
   cmin=[cmin;x(:,m) y];
end
x=[37.5:-0.1:36.9755];
for n=1:length(x)
   y=-sqrt(r^2-x(:,n)^2);
```

```
  cmin=[cmin;x(:,n) y];
end
%=========================================================
% Define ray (footpoint and desired direction of travel)
thetain=input('Desired_Direction of travel: ');
if (thetain==90) | (thetain==-270)
 thetar=-89.5*pi/180;
elseif (thetain==-90) | (thetain==270)
 thetar=89.5*pi/180;
else
 thetar=-thetain*pi/180;
end
xray=input('Enter known footpoint, X coordinate: ');
yray=input('Enter known footpoint, Y coordinate: ');
%=========================================================
% Determine if new footpoint is within pos & neg leg limits
distLpn=(ylinen-yray)*cos(thetan)-(xlinen-xray)*sin(thetan);
distLpp=(ylinep-yray)*cos(thetap)-(xlinep-xray)*sin(thetap);
%=========================================================
if (((distLpn>0) | (distLpn==0)) & ((distLpp<0) | (distLpp==0)))
 % Determine whether there is a positive intersection
 thetapos=thetar-thetap;
 if (thetapos==0) | (thetapos==pi) | (thetapos==-pi)
  disp('Desired_Direction and Positive_Leg_Limit are parallel.')
  xintpos=xray;
  yintpos=yray;
 else
  b=(xlinep*sin(thetap))-(ylinep*cos(thetap));
  a=(xray*sin(thetar))-(yray*cos(thetar));
  xtop=(-cos(thetap)*a)+(cos(thetar)*b);
  ytop=(sin(thetar)*b)-(sin(thetap)*a);
  bottom=sin(thetap-thetar);
  xintpos=xtop/bottom;
  yintpos=ytop/bottom;
  % Determine if the ray intersects the pos line segment
  Lp=(xlinep*cos(thetap)+ylinep*sin(thetap));
  Rp=(Rxp*cos(thetap)+Ryp*sin(thetap));
  segtest=(xintpos*cos(thetap)+yintpos*sin(thetap));
  if ((Lp<segtest) & (Rp>segtest))  % if true, intersection
   % Determine if the orientation is correct
   p2=(xintpos*cos(thetar)+yintpos*sin(thetar));
   p1=(xray*cos(thetar)+yray*sin(thetar));
   if p2>p1  % correct orientation
    Flag=1;
   end
  end
 end
%=========================================================
 % Determine whether there is a negative intersection
 thetaneg=thetar-thetan;
 if (thetaneg==0) | (thetaneg==pi) | (thetaneg==-pi)
  disp('Desired_Direction and Negative_Leg_Limit are parallel.')
  xintneg=xray;
  yintneg=yray;
 else
  b=(xlinen*sin(thetan))-(ylinen*cos(thetan));
  a=(xray*sin(thetar))-(yray*cos(thetar));
  xtop=(-cos(thetan)*a)+(cos(thetar)*b);
  ytop=(sin(thetar)*b)-(sin(thetan)*a);
  bottom=sin(thetan-thetar);
  xintneg=xtop/bottom;
  yintneg=ytop/bottom;
  % Determine if the ray intersects the neg line segment
  Ln=(xlinen*cos(thetan)+ylinen*sin(thetan));
  Rn=(Rxn*cos(thetan)+Ryn*sin(thetan));
  segtest=(xintneg*cos(thetan)+yintneg*sin(thetan));
  if ((Ln<segtest) & (Rn>segtest))  % if true, intersection
```

```
    % Determine if the orientation is correct
    p2=(xintneg*cos(thetar)+yintneg*sin(thetar));
    p1=(xray*cos(thetar)+yray*sin(thetar));
    if p2>p1  % correct orientation
     Flag=-1;
    end
   end
  end
%=============================================================
 % Calculate the line segments
 posint=[];
 x=xlinep:1:160;
 for p=1:length(x)
  y=(-0.5774*x(:,p))+14.4338;
  posint=[posint;x(:,p) y];
 end
 negint=[];
 x=xlinen:1:160;
 for q=1:length(x)
  y=(0.577*x(:,q))-14.4338;
  negint=[negint;x(:,q) y];
 end
%=============================================================
% Draw the orientation of the ray
 ray=[];
 if Flag==1  % positive intersection is good
  if xintpos<xray
   x=xintpos:1:xray;
  else
   x=xray:1:xintpos;
  end
  m=(yray-yintpos)/(xray-xintpos);
  b=yray-(m*xray);
 elseif Flag==-1  % negative intersection is good
  if xintneg<xray
   x=xintneg:1:xray;
  else
   x=xray:1:xintneg;
  end
  m=(yintneg-yray)/(xintneg-xray);
  b=yray-(m*xray);
 else
  Flag=0;  % no intersection with line segments


  % +++++++insert intersection with arcs here+++++++


 end
 for g=1:length(x)
  y=(m*x(:,g))+b;
  ray=[ray;x(:,g) y];
 end
%=============================================================
 % Plot the results
 axis([0 200 -100 100])
 axis('square')
 plot(posint(:,1),posint(:,2),'g')
 hold on
 plot(negint(:,1),negint(:,2),'g')
 plot(cmin(:,1),cmin(:,2),'g')
 plot(cmax(:,1),cmax(:,2),'g')
 grid
 plot(xray,yray,'+c5')
 if Flag==1
  plot(xintpos,yintpos,'or')
 elseif Flag==-1
```

```
  plot(xintneg,yintneg,'or')
 else
  Flag=0;
  text(0.25,0.2,'Ray intersects only arcs.','sc')
 end
 plot(ray(:,1),ray(:,2),'b')
 title('Intersection of Direction and Workspace')
 xlabel('X coordinates, (cm)')
 ylabel('Y coordinates, (cm)')
 hold off
else   % new footpoint not inside pos & neg leg limits
 % Calculate the line segments
 posint=[];
 x=xlinep:1:160;
 for p=1:length(x)
  y=(-0.5774*x(:,p))+14.4338;
  posint=[posint;x(:,p) y];
 end
 negint=[];
 x=xlinen:1:160;
 for q=1:length(x)
  y=(0.577*x(:,q))-14.4338;
  negint=[negint;x(:,q) y];
 end
%=================================================================
 % Plot the results
 axis([0 200 -100 100])
 axis('square')
 plot(posint(:,1),posint(:,2),'g')
 hold on
 plot(negint(:,1),negint(:,2),'g')
 plot(cmin(:,1),cmin(:,2),'g')
 plot(cmax(:,1),cmax(:,2),'g')
 grid
 plot(xray,yray,'+c5')
 title('Intersection of Direction and Workspace')
 xlabel('X coordinates, (cm)')
 ylabel('Y coordinates, (cm)')
 text(0.25,0.2,'Footpoint not inside limits.','sc')
 hold off
end

% ************************************************************************
% Title:   segint.m
% Inputs:  (1,2,3,4,5)
% Outputs:  [1,2,3]
% Purpose:  This Matlab function determines the intersection of a ray
%         and a line segment.
% ************************************************************************
%
function [intcpt,dist,flag] = segint(foot, direc, inseg, outseg, vector)
%
% intcpt = (x,y) coordinates of intercept
% dist = distance between foot and intercepts
% flag = 1 if intercept found; flag = 0 if no intercept found
% foot = (x,y) foot components
% direc = direction of interest
% inseg/outseg = endpoints of segment
% vector = 1, line segment is clockwise
% vector = 0, line segment is counter-clockwise
%
limit = 215; % maximum possible distance
flag = 0;
direc = direc*pi/180;
%
% Find theta of segment
thetaseg = atan2(outseg(1,2) - inseg(1,2), outseg(1,1) - inseg(1,1));
```

```
%
% Determine if foot falls on the line segment
distLp = ( ((foot(1,2)-inseg(1,2))*cos(thetaseg)) - ((foot(1,1)-inseg(1,1))*sin(thetaseg)) );
distLp = round(distLp);
if (distLp == 0)
 flag = 1; % foot is on line segment; quit
 dist = limit;
 intcpt = [foot(1,1),foot(1,2)];
elseif ( (distLp > 0) & (vector == 0) )
 flag = -1; % foot is outside workspace; quit
 dist = limit;
 intcpt = [];
elseif ( (distLp < 0) & (vector == 0) )
 flag = 0; % foot is inside workspace; find intersection
elseif ( (distLp < 0) & (vector == 1) )
 flag = -1; % foot is outside workspace; quit
 dist = limit;
 intcpt = [];
elseif ( (distLp > 0) & (vector == 1) )
 flag = 0; % foot is inside workspace; find intersection
end
%
if (flag == 0)
 thetadif = direc - thetaseg;
 if (thetadif ~= 0) & (thetadif ~= pi) & (thetadif ~= -pi)
  b = (inseg(1,1)*sin(thetaseg)) - (inseg(1,2)*cos(thetaseg));
  a = (foot(1,1)*sin(direc)) - (foot(1,2)*cos(direc));
  xtop = (-cos(thetaseg)*a) + (cos(direc)*b);
  ytop = (sin(direc)*b) - (sin(thetaseg)*a);
  bottom = sin(thetaseg - direc);
  xint = xtop/bottom;
  yint = ytop/bottom;
  intcpt = [xint, yint];
  % Determine if the ray intersects the line segment
  CCW = (inseg(1,1)*cos(thetaseg) + inseg(1,2)*sin(thetaseg));
  CW = (outseg(1,1)*cos(thetaseg) + outseg(1,2)*sin(thetaseg));
  segtest = (xint*cos(thetaseg) + yint*sin(thetaseg));
  if ((CCW <= segtest) & (CW >= segtest))  % if true, intersection
   % Determine if the orientation is correct
   p2 = (xint*cos(direc) + yint*sin(direc));
   p1 = (foot(1,1)*cos(direc) + foot(1,2)*sin(direc));
   if p2 > p1  % correct orientation
    flag = 1;
    dist = sqrt((yint - foot(1,2))^2 + (xint - foot(1,1))^2);
   else
    flag = 0;
   end
  end
 end
end
%
if ( (flag == -1) | (flag == 0) )
 flag = 0;
 dist = limit;
 intcpt = [];
end

% ************************************************************************
% Title:   segint1.m
% Inputs:  (1,2,3,4)
% Outputs:  [1,2,3]
% Purpose:  This Matlab function determines the intersection of a ray
%         and a line segment.
% ************************************************************************
%
function [intcpt,dist,flag] = segint1(foot, direc, inseg, outseg)
%
```

```
% intcpt = (x,y) coordinates of intercept
% dist = distance between foot and intercepts
% flag = 1 if intercept found; flag = 0 if no intercept found
% foot = (x,y) foot components
% direc = direction of interest
% inseg/outseg = endpoints of segment
%
flag = 0;
direc = direc*pi/180;
%
% Find theta of segment
thetaseg = atan2(outseg(1,2) - inseg(1,2), outseg(1,1) - inseg(1,1))
%
 % Determine if the foot is on the line segment
 CCW = (inseg(1,1)*cos(thetaseg) + inseg(1,2)*sin(thetaseg))
 CW = (outseg(1,1)*cos(thetaseg) + outseg(1,2)*sin(thetaseg))
 segtest = (foot(1,1)*cos(thetaseg) + foot(1,2)*sin(thetaseg))
 pause
 if ((CCW <= segtest) & (CW >= segtest))  % if true, foot is on line
  flag = 1;
  dist = 0;
  intcpt = foot;
 end
if (flag == 0)
 thetadif = direc - thetaseg;
 if (thetadif ~= 0) & (thetadif ~= pi) & (thetadif ~= -pi)
  b = (inseg(1,1)*sin(thetaseg)) - (inseg(1,2)*cos(thetaseg));
  a = (foot(1,1)*sin(direc)) - (foot(1,2)*cos(direc));
  xtop = (-cos(thetaseg)*a) + (cos(direc)*b);
  ytop = (sin(direc)*b) - (sin(thetaseg)*a);
  bottom = sin(thetaseg - direc);
  xint = xtop/bottom;
  yint = ytop/bottom;
  intcpt = [xint, yint];
  % Determine if the ray intersects the line segment
  CCW = (inseg(1,1)*cos(thetaseg) + inseg(1,2)*sin(thetaseg));
  CW = (outseg(1,1)*cos(thetaseg) + outseg(1,2)*sin(thetaseg));
  segtest = (xint*cos(thetaseg) + yint*sin(thetaseg));
  if ((CCW <= segtest) & (CW >= segtest))  % if true, intersection
   % Determine if the orientation is correct
   p2 = (xint*cos(direc) + yint*sin(direc));
   p1 = (foot(1,1)*cos(direc) + foot(1,2)*sin(direc));
   if p2 > p1  % correct orientation
    flag = 1;
    dist = sqrt((yint - foot(1,2))^2 + (xint - foot(1,1))^2);
   else
    flag = 0;
   end
  end
 end
end
if (flag ~= 1)
 flag = 0;
 dist = [];
 intcpt = [];
end

% **********.  '**************************************************
% Title:   stabint.m
% Inputs:  (1,2,3,4,5,6,7,8,9,10)
% Outputs:  [1,2,3,4]
% Purpose:  This Matlab function determines the intersection of a ray
%         and a line segment in one Tripod.
% ************************************************************************
%
function [xint,yint,dist,flag] = stabint(T,rx,ry,rtheta,ax,ay,bx,by,cx,cy)
%
```

```
% T = tripod number, 0 or 1
% ax/y, bx/y, cx/y = selected tripod foot positions
% rx/ry = ray components
% rtheta = direction of interest
% xint/yint = segment intercepts
% dist = distance between rx/ry and xint/yint
% flag = 1 if intercept found; flag = 0 if no intercept found
% lega = leg1/2, legb = leg3/4, legc = leg5/6
%
% convert degrees to radians and
% correct for differences in positive angle convention used
% in computer and used on AquaRobot; AquaRobot positive
% angle is CW
rtheta = -(rtheta*pi/180);
%
% correct that +y axis in Matlab is -y axis in AquaRobot
ry = -ry;
ay = -ay;
by = -by;
cy = -cy;
%
flag = 0;
if T==0
%=========================
% test segment 1-5 (lega-legc)
%
% Find theta of segment
thetaseg=atan2(ay-cy,ax-cx);
%
thetadif=rtheta-thetaseg;
if (thetadif==0) | (thetadif==pi) | (thetadif==-pi)
 disp('parallel') % desired direction and line segment are parallel
else
 b=(cx*sin(thetaseg))-(cy*cos(thetaseg));
 a=(rx*sin(rtheta))-(ry*cos(rtheta));
 xtop=(-cos(thetaseg)*a)+(cos(rtheta)*b);
 ytop=(sin(rtheta)*b)-(sin(thetaseg)*a);
 bottom=sin(thetaseg-rtheta);
 x15int=xtop/bottom;
 y15int=ytop/bottom;
 % Determine if the ray intersects the line segment
 legcpt=(cx*cos(thetaseg)+cy*sin(thetaseg));
 legapt=(ax*cos(thetaseg)+ay*sin(thetaseg));
 segtest=(x15int*cos(thetaseg)+y15int*sin(thetaseg));
 if ((legcpt<=segtest) & (segtest<=legapt))  % if true, intersection
  % Determine if the orientation is correct
  p2=(x15int*cos(rtheta)+y15int*sin(rtheta));
  p1=(rx*cos(rtheta)+ry*sin(rtheta));
  if p2>p1  % correct orientation
   dist=sqrt((y15int-ry)^2+(x15int-rx)^2);
   xint = x15int;
   yint = -y15int; % recorrect y axis
  end
 end
end
%=========================
% test segment 5-3 (legc-legb)
%
% Find theta of segment
thetaseg=atan2(cy-by,cx-bx);
%
thetadif=rtheta-thetaseg;
if (thetadif==0) | (thetadif==pi) | (thetadif==-pi)
 disp('parallel') % desired direction and line segment are parallel
else
 b=(bx*sin(thetaseg))-(by*cos(thetaseg));
 a=(rx*sin(rtheta))-(ry*cos(rtheta));
```

```
 xtop=(-cos(thetaseg)*a)+(cos(rtheta)*b);
 ytop=(sin(rtheta)*b)-(sin(thetaseg)*a);
 bottom=sin(thetaseg-rtheta);
 x53int=xtop/bottom;
 y53int=ytop/bottom;
 % Determine if the ray intersects the line segment
 legcpt=(cx*cos(thetaseg)+cy*sin(thetaseg));
 legbpt=(bx*cos(thetaseg)+by*sin(thetaseg));
 segtest=(x53int*cos(thetaseg)+y53int*sin(thetaseg));
 if ((legcpt>=segtest) & (segtest>=legbpt))  % if true, intersection
  % Determine if the orientation is correct
  p2=(x53int*cos(rtheta)+y53int*sin(rtheta));
  p1=(rx*cos(rtheta)+ry*sin(rtheta));
  if p2>p1  % correct orientation
   dist=sqrt((y53int-ry)^2+(x53int-rx)^2);
   xint = x53int;
   yint = -y53int; % recorrect y axis
  end
 end
end
%=====================
% test segment 3-1 (legb-lega)
%
% Find theta of segment
thetaseg=atan2(ay-by,ax-bx);
%
thetadif=rtheta-thetaseg;
if (thetadif==0) | (thetadif==pi) | (thetadif==-pi)
 disp('parallel') % desired direction and line segment are parallel
else
 b=(bx*sin(thetaseg))-(by*cos(thetaseg));
 a=(rx*sin(rtheta))-(ry*cos(rtheta));
 xtop=(-cos(thetaseg)*a)+(cos(rtheta)*b);
 ytop=(sin(rtheta)*b)-(sin(thetaseg)*a);
 bottom=sin(thetaseg-rtheta);
 x31int=xtop/bottom;
 y31int=ytop/bottom;
 % Determine if the ray intersects the line segment
 legapt=(ax*cos(thetaseg)+ay*sin(thetaseg));
 legbpt=(bx*cos(thetaseg)+by*sin(thetaseg));
 segtest=(x31int*cos(thetaseg)+y31int*sin(thetaseg));
 if ((legbpt<=segtest) & (segtest<=legapt))  % if true, intersection
  % Determine if the orientation is correct
  p2=(x31int*cos(rtheta)+y31int*sin(rtheta));
  p1=(rx*cos(rtheta)+ry*sin(rtheta));
  if p2>p1  % correct orientation
   dist=sqrt((y31int-ry)^2+(x31int-rx)^2);
   xint = x31int;
   yint = -y31int; % recorrect y axis
  end
 end
end
%
elseif T==1
%====================
% test segment 2-6 (lega-legc)
%
% Find theta of segment
thetaseg=atan2(ay-cy,ax-cx);
%
thetadif=rtheta-thetaseg;
if (thetadif==0) | (thetadif==pi) | (thetadif==-pi)
 disp('parallel') % desired direction and line segment are parallel
else
 b=(cx*sin(thetaseg))-(cy*cos(thetaseg));
 a=(rx*sin(rtheta))-(ry*cos(rtheta));
 xtop=(-cos(thetaseg)*a)+(cos(rtheta)*b);
```

```
 ytop=(sin(rtheta)*b)-(sin(thetaseg)*a);
 bottom=sin(thetaseg-rtheta);
 x26int=xtop/bottom;
 y26int=ytop/bottom;
 % Determine if the ray intersects the line segment
 legapt=(ax*cos(thetaseg)+ay*sin(thetaseg));
 legcpt=(cx*cos(thetaseg)+cy*sin(thetaseg));
 segtest=(x26int*cos(thetaseg)+y26int*sin(thetaseg));
 if ((legcpt<=segtest) & (segtest<=legapt))  % if true, intersection
  % Determine if the orientation is correct
  p2=(x26int*cos(rtheta)+y26int*sin(rtheta));
  p1=(rx*cos(rtheta)+ry*sin(rtheta));
  if p2>p1  % correct orientation
   dist=sqrt((y26int-ry)^2+(x26int-rx)^2);
   xint = x26int;
   yint = -y26int; % recorrect y axis
  end
 end
end
%
%=====================
% test segment 6-4 (legc-legb)
%
% Find theta of segment
thetaseg=atan2(cy-by,cx-bx);
%
thetadif=rtheta-thetaseg;
if (thetadif==0) | (thetadif==pi) | (thetadif==-pi)
 disp('parallel') % desired direction and line segment are parallel
else
 b=(bx*sin(thetaseg))-(by*cos(thetaseg));
 a=(rx*sin(rtheta))-(ry*cos(rtheta));
 xtop=(-cos(thetaseg)*a)+(cos(rtheta)*b);
 ytop=(sin(rtheta)*b)-(sin(thetaseg)*a);
 bottom=sin(thetaseg-rtheta);
 x64int=xtop/bottom;
 y64int=ytop/bottom;
 % Determine if the ray intersects the line segment
 legcpt=(cx*cos(thetaseg)+cy*sin(thetaseg));
 legbpt=(bx*cos(thetaseg)+by*sin(thetaseg));
 segtest=(x64int*cos(thetaseg)+y64int*sin(thetaseg));
 if ((legcpt<=segtest) & (segtest<=legbpt))  % if true, intersection
  % Determine if the orientation is correct
  p2=(x64int*cos(rtheta)+y64int*sin(rtheta));
  p1=(rx*cos(rtheta)+ry*sin(rtheta));
  if p2>p1  % correct orientation
   dist=sqrt((y64int-ry)^2+(x64int-rx)^2);
   xint = x64int;
   yint = -y64int; % recorrect y axis
  end
 end
end
%
%=====================
% test segment 4-2 (legb-lega)
%
% Find theta of segment
thetaseg=atan2(by-ay,bx-ax);
%
thetadif=rtheta-thetaseg;
if (thetadif==0) | (thetadif==pi) | (thetadif==-pi)
 disp('parallel') % desired direction and line segment are parallel
else
 b=(ax*sin(thetaseg))-(ay*cos(thetaseg));
 a=(rx*sin(rtheta))-(ry*cos(rtheta));
 xtop=(-cos(thetaseg)*a)+(cos(rtheta)*b);
 ytop=(sin(rtheta)*b)-(sin(thetaseg)*a);
```

```
 bottom=sin(thetaseg-rtheta);
 x42int=xtop/bottom;
 y42int=ytop/bottom;
 % Determine if the ray intersects the line segment
 legapt=(ax*cos(thetaseg)+ay*sin(thetaseg));
 legbpt=(bx*cos(thetaseg)+by*sin(thetaseg));
 segtest=(x42int*cos(thetaseg)+y42int*sin(thetaseg));
 if ((legapt<=segtest) & (segtest<=legbpt))  % if true, intersection
  % Determine if the orientation is correct
  p2=(x42int*cos(rtheta)+y42int*sin(rtheta));
  p1=(rx*cos(rtheta)+ry*sin(rtheta));
  if p2>p1  % correct orientation
   dist=sqrt((y42int-ry)^2+(x42int-rx)^2);
   xint = x42int;
   yint = -y42int; % recorrect y axis
  end
 end
end
%
else
 disp('Incorrect Tripod number')
end
if dist==[]
 flag=0;
else
 flag=1;
end

% ************************************************************************
% Title:   stable.m
% Inputs:  (1,2,3,4,5,6,7,8)
% Outputs:  [1,2]
% Purpose:  This Matlab function calculates the stability of a Tripod.
% ************************************************************************
%
function [flag,sm]=stable(CBx,CBy,legax,legay,legbx,legby,legcx,legcy)
% CB,lega,legb,legc oriented CCW so S is positive when CB is inside.
%
% 'flag' indicates degee of stability
% 'sm' is the stability margin
% lega=leg1/2, legb=leg3/4, legc=leg5/6
x1 = CBx;
y1 = -CBy; %+y axis in Matlab is -y axis in AquaRobot
x2 = legax;
y2 = -legay; %+y axis in Matlab is -y axis in AquaRobot
x3 = legbx;
y3 = -legby; %+y axis in Matlab is -y axis in AquaRobot
x4 = legcx;
y4 = -legcy; %+y axis in Matlab is -y axis in AquaRobot
%
L3 = sqrt((x3-x2)^2+(y3-y2)^2);
L2 = sqrt((x4-x3)^2+(y4-y3)^2);
L1 = sqrt((x2-x4)^2+(y2-y4)^2);
%
S1 = ((x2-x1)*(y4-y1)-(x4-x1)*(y2-y1))/2;
S2 = ((x4-x1)*(y3-y1)-(x3-x1)*(y4-y1))/2;
S3 = ((x3-x1)*(y2-y1)-(x2-x1)*(y3-y1))/2;
%
if (S1>0) & (S2>0) & (S3>0)
 flag=1;  % The new tripod is stable.
elseif (S1<0) | (S2<0) | (S3<0)
 flag=-1; % The new tripod is unstable.
else
 flag=0;  % The new tripod is marginally stable
end
%
H1=2*S1/L1;
```

```
H2=2*S2/L2;
H3=2*S3/L3;
%
sm=min([H1 H2 H3]);

% ************************************************************************
% Title:   start.m
% Inputs:
% Outputs:  kcS.met plot; ks data file
% Purpose:  This Matlab program plots the START posture for AquaRobot.
% ************************************************************************
%
!del ks
!del kcS.met
clear
hold off
%
angle01=0*pi/180;  % leg1
angle02=60*pi/180;  % leg2
angle03=120*pi/180; % leg3
angle04=180*pi/180; % leg4
angle05=240*pi/180; % leg5
angle06=300*pi/180; % leg6
%
a0=37.5;
a1=20;
a2=50;
a3=100;
%
angle1 = 0;
%
angle2F = -35.8585*pi/180;
angle2B = 35.8585*pi/180;
%
angle3F = 125.8585*pi/180;
angle3B = -125.8585*pi/180;
%
T00 = [cos(angle01) -sin(angle01) 0 0;sin(angle01) cos(angle01) 0 0;0 0 1 0;0 0 0 1];
T01 = [cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP = T00*T01;
T12 = [cos(angle2F) -sin(angle2F) 0 a1;0 0 1 0;-sin(angle2F) -cos(angle2F) 0 0;0 0 0 1];
KNEE1 = T00*T01*T12;
T23 = [cos(angle3F) -sin(angle3F) 0 a2;sin(angle3F) cos(angle3F) 0 0;0 0 1 0;0 0 0 1];
KNEE2 = T00*T01*T12*T23;
T34 = [1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT1 = T00*T01*T12*T23*T34;
diary off
%
T00 = [cos(angle02) -sin(angle02) 0 0;sin(angle02) cos(angle02) 0 0;0 0 1 0;0 0 0 1];
T01 = [cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP = T00*T01;
T12 = [cos(angle2F) -sin(angle2F) 0 a1;0 0 1 0;-sin(angle2F) -cos(angle2F) 0 0;0 0 0 1];
KNEE1 = T00*T01*T12;
T23 = [cos(angle3F) -sin(angle3F) 0 a2;sin(angle3F) cos(angle3F) 0 0;0 0 1 0;0 0 0 1];
KNEE2 = T00*T01*T12*T23;
T34 = [1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT2 = T00*T01*T12*T23*T34;
diary off
%
T00 = [cos(angle03) -sin(angle03) 0 0;sin(angle03) cos(angle03) 0 0;0 0 1 0;0 0 0 1];
T01 = [cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP = T00*T01;
T12 = [cos(angle2B) -sin(angle2B) 0 a1;0 0 1 0;-sin(angle2B) -cos(angle2B) 0 0;0 0 0 1];
KNEE1 = T00*T01*T12;
T23 = [cos(angle3B) -sin(angle3B) 0 a2;sin(angle3B) cos(angle3B) 0 0;0 0 1 0;0 0 0 1];
```

```
KNEE2 = T00*T01*T12*T23;
T34 = [1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT3 = T00*T01*T12*T23*T34;
diary off
%
T00 = [cos(angle04) -sin(angle04) 0 0;sin(angle04) cos(angle04) 0 0;0 0 1 0;0 0 0 1];
T01 = [cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP = T00*T01;
T12 = [cos(angle2B) -sin(angle2B) 0 a1;0 0 1 0;-sin(angle2B) -cos(angle2B) 0 0;0 0 0 1];
KNEE1 = T00*T01*T12;
T23 = [cos(angle3B) -sin(angle3B) 0 a2;sin(angle3B) cos(angle3B) 0 0;0 0 1 0;0 0 0 1];
KNEE2 = T00*T01*T12*T23;
T34 = [1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT4 = T00*T01*T12*T23*T34;
diary off
%
T00 = [cos(angle05) -sin(angle05) 0 0;sin(angle05) cos(angle05) 0 0;0 0 1 0;0 0 0 1];
T01 = [cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP = T00*T01;
T12 = [cos(angle2B) -sin(angle2B) 0 a1;0 0 1 0;-sin(angle2B) -cos(angle2B) 0 0;0 0 0 1];
KNEE1 = T00*T01*T12;
T23 = [cos(angle3B) -sin(angle3B) 0 a2;sin(angle3B) cos(angle3B) 0 0;0 0 1 0;0 0 0 1];
KNEE2 = T00*T01*T12*T23;
T34 = [1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT5 = T00*T01*T12*T23*T34;
diary off
%
T00 = [cos(angle06) -sin(angle06) 0 0;sin(angle06) cos(angle06) 0 0;0 0 1 0;0 0 0 1];
T01 = [cos(angle1) -sin(angle1) 0 a0;sin(angle1) cos(angle1) 0 0;0 0 1 0;0 0 0 1];
HIP = T00*T01;
T12 = [cos(angle2F) -sin(angle2F) 0 a1;0 0 1 0;-sin(angle2F) -cos(angle2F) 0 0;0 0 0 1];
KNEE1 = T00*T01*T12;
T23 = [cos(angle3F) -sin(angle3F) 0 a2;sin(angle3F) cos(angle3F) 0 0;0 0 1 0;0 0 0 1];
KNEE2 = T00*T01*T12*T23;
T34 = [1 0 0 a3;0 1 0 0;0 0 1 0;0 0 0 1];
diary ks
FOOT6 = T00*T01*T12*T23*T34;
diary off
%
x = [FOOT1(1,4) FOOT2(1,4) FOOT3(1,4) FOOT4(1,4) FOOT5(1,4) FOOT6(1,4)];
y = [FOOT1(2,4) FOOT2(2,4) FOOT3(2,4) FOOT4(2,4) FOOT5(2,4) FOOT6(2,4)];
axis([-200 200 -200 200])
plot(x,y,'o')
hold
x = 0;
y = 0;
plot(x,y,'+g')
xlabel('X Foot Coordinates (cm)')
ylabel('Y Foot Coordinates (cm)')
title('AquaRobot START Posture')
grid
gtext('FOOT1')
gtext('FOOT2')
gtext('Body Center')
meta kcS

% ************************************************************************
% Title:   ucwlink1.m
% Inputs:
% Outputs:  c4.met plot
% Purpose:  This Matlab program plots the AVAILABLE workspace of
%         the HIP (Joint 1) to KNEE1 (Joint 2) LINK1 of the AquaRobot.
% ************************************************************************
%
```

```
!del c4.met
angle0=0*pi/180;
angle2=0*pi/180;
a0=37.5;
a1=20;
%
a1vec=[];
for angle1=(-60*pi/180):(5*pi/180):(60*pi/180)
    TB0=[cos(angle0) -sin(angle0) 0 0;
       sin(angle0) cos(angle0) 0 0;
       0 0 1 0;
       0 0 0 1];
%
    T01=[cos(angle1) -sin(angle1) 0 a0;
       sin(angle1) cos(angle1) 0 0;
       0 0 1 0;
       0 0 0 1];
%
    T12=[cos(angle2) -sin(angle2) 0 a1;
       0 0 1 0;
       -sin(angle2) -cos(angle2) 0 0;
       0 0 0 1];
%
    LINK1=TB0*T01*T12;
    a1vec=[a1vec;LINK1(1,4) LINK1(2,4)];
end
axis([0 60 -30 30])
plot(a1vec(:,1),a1vec(:,2))
grid
title('Hip Workspace')
xlabel('X coordinate, cm')
ylabel('Y coordinate, cm')
gtext('HIP')
gtext('-60 degrees')
gtext('60 degrees')
gtext('KNEE1')
meta c4

% ************************************************************************
% Title:   ucwlink2.m
% Inputs:
% Outputs:  c5.met plot
% Purpose:  This Matlab program plots the AVAILABLE workspace of
%        the KNEE1 (Joint 2) to KNEE2 (Joint 3) LINK2 of the AquaRobot.
% ************************************************************************
%
!del c5.met
%
angle0=0*pi/180;
angle1=0*pi/180;
angle3=0*pi/180;
a0=37.5;
a1=20;
a2=50;
%
a2vec=[];
for angle2=(-73.4*pi/180):(5*pi/180):(106.6*pi/180)
    TB0=[cos(angle0) -sin(angle0) 0 0;
       sin(angle0) cos(angle0) 0 0;
       0 0 1 0;
       0 0 0 1];
%
    T01=[cos(angle1) -sin(angle1) 0 a0;
       sin(angle1) cos(angle1) 0 0;
       0 0 1 0;
       0 0 0 1];
%
```

```
    T12=[cos(angle2) -sin(angle2) 0 a1;
       0 0 1 0;
       -sin(angle2) -cos(angle2) 0 0;
       0 0 0 1];
%
    T23=[cos(angle3) -sin(angle3) 0 a2;
      sin(angle3) cos(angle3) 0 0;
      0 0 1 0;
      0 0 0 1];
%
    LINK2=TB0*T01*T12*T23;
    a2vec=[a2vec;LINK2(1,4) LINK2(3,4)];
end
axis([0 120 -60 60])
plot(a2vec(:,1),a2vec(:,2))
grid
title('Knee1 Workspace')
xlabel('X coordinate, cm')
ylabel('Z coordinate, cm')
gtext('KNEE1')
gtext('73.4 degrees')
gtext('-106.6 degrees')
gtext('KNEE2')
meta c5

% ************************************************************************
% Title:   ucwlink3.m
% Inputs:
% Outputs:  c6.met plot
% Purpose:  This Matlab program plots the AVAILABLE workspace of
%        the KNEE2 (Joint 3) to FOOT (Joint 4) LINK3 of the AquaRobot.
% ************************************************************************
%
!del c6.met
%
angle0=0*pi/180;
angle1=0*pi/180;
angle2=0*pi/180;
a0=37.5;
a1=20;
a2=50;
a3=100;
%
a3vec=[];
for angle3=(-23.6*pi/180):(5*pi/180):(156.4*pi/180)
    TB0=[cos(angle0) -sin(angle0) 0 0;
       sin(angle0) cos(angle0) 0 0;
       0 0 1 0;
       0 0 0 1];
%
    T01=[cos(angle1) -sin(angle1) 0 a0;
       sin(angle1) cos(angle1) 0 0;
       0 0 1 0;
       0 0 0 1];
%
    T12=[cos(angle2) -sin(angle2) 0 a1;
       0 0 1 0;
       -sin(angle2) -cos(angle2) 0 0;
       0 0 0 1];
%
    T23=[cos(angle3) -sin(angle3) 0 a2;
      sin(angle3) cos(angle3) 0 0;
      0 0 1 0;
      0 0 0 1];
%
    T34=[1 0 0 a3;
      0 1 0 0;
```

```
        0 0 1 0;
        0 0 0 1];
%
    LINK3=TB0*T01*T12*T23*T34;
    a3vec=[a3vec;LINK3(1,4) LINK3(3,4)];
end
axis([0 250 -125 125])
plot(a3vec(:,1),a3vec(:,2))
grid
title('Knee2 Workspace')
xlabel('X coordinate, cm')
ylabel('Z coordinate, cm')
gtext('KNEE2')
gtext('23.6 degrees')
gtext('-156.4 degrees')
gtext('FOOT')
meta o6

% ************************************************************************
% Title:   w2btrans.m
% Inputs:  (1,2,3,4,5,6,7,8,9)
% Outputs:  [1]
% Purpose:  This function transforms world coordinates to body coordinates.
% ************************************************************************
%
function [body] = w2btrans(phi, theta, psi, xtrans, ytrans, ztrans, wx, wy, wz)
%
% phi = rotation of {B} about the Xw-axis;
% theta = rotation of {B} about the Yw-axis;
% psi = rotation of {B} about the Zw-axis;
% xtrans = X-axis translation of {B} with respect to {W};
% ytrans = Y-axis translation of {B} with respect to {W};
% ztrans = Z-axis translation of {B} with respect to {W};
% wx = known Xworld coordinate;
% wy = known Yworld coordinate;
% wz = known Zworld coordinate;
%
% construct the translation vector, the body vector, and the rotation matrix
trans = [xtrans, ytrans, ztrans]';
world = [wx, wy,-wz, 1]';
rotate = [cos(psi)*cos(theta) cos(psi)*sin(theta)*sin(phi)-sin(psi)*cos(phi) cos(psi)*sin(theta)*cos(phi)+sin(psi)*sin(phi);
      sin(psi)*cos(theta) sin(psi)*sin(theta)*sin(phi)+cos(psi)*cos(phi) sin(psi)*sin(theta)*cos(phi)-cos(psi)*sin(phi);
      -sin(theta) cos(theta)*sin(phi) cos(theta)*cos(phi)];
%
% build the body-to-world homogeneous matrix
BWT = [rotate trans;0 0 0 1];
%
% find the resultant 4X1 vector of equivalent body coordinates
Tbody = inv(BWT)*world;
%
% strip off the (x,y,z) coordinates
body = [Tbody(1,1), Tbody(2,1), Tbody(3,1)];

% ************************************************************************
% Title:   work.m
% Inputs:
% Outputs:  work.met plot
% Purpose:  This Matlab program draws the UNCONSTRAINED workspace for
%       AquaRobot.
% ************************************************************************
%
!del work.met
clg
clc
clear
hold off
%=======================================
```

```
% construct body with radius 37.5 cm
c=[];
r=37.5;
x=[-37.5:0.5:37.5];
for m=1:length(x)
   y=sqrt(r^2-x(:,m)^2);
   c=[c;x(:,m) y];
end
x=[37:-0.5:-37.5];
for n=1:length(x)
   y=-sqrt(r^2-x(:,n)^2);
   c=[c;x(:,n) y];
end
axis('square')
axis([-200 200 -200 200])
plot(c(:,1),c(:,2))
grid
hold
%=============================================
% construct leg1 2D workspace
d=[];
x=37.5:0.5:200;
for p=1:length(x)
  y=(1.73205*x(:,p))-64.9519;
  d=[d;x(:,p) y];
end
plot(d(:,1),d(:,2),'g')
%
e=[];
x=37.5:0.5:200;
for q=1:length(x)
  y=(-1.73205*x(:,q))+64.9519;
  e=[e;x(:,q) y];
end
plot(e(:,1),e(:,2),'g')
%
% show leg1
L1=[];
x=37.5:0.5:178.2107;
for q=1:length(x)
  y=(0*x(:,q))+0;
  L1=[L1;x(:,q) y];
end
plot(L1(:,1),L1(:,2),'b')
%
% show foot1
x=178.2107;
y=0;
plot(x,y,'oc5')
%=============================================
% construct leg4 2D workspace
f=[];
x=-180:0.5:-37.5;
for r=1:length(x)
  y=(-1.73205*x(:,r))-64.9519;
  f=[f;x(:,r) y];
end
plot(f(:,1),f(:,2),'g')
%
g=[];
x=-200:0.5:-37.5;
for s=1:length(x)
  y=(1.73205*x(:,s))+64.9519;
  g=[g;x(:,s) y];
end
plot(g(:,1),g(:,2),'g')
%
```

```
% show leg4
L4=[];
x=-37.5:-0.5:-178.2107;
for q=1:length(x)
  y=(0*x(:,q))+0;
  L4=[L4;x(:,q) y];
end
plot(L4(:,1),L4(:,2),'b')
%
% show foot4
x=-178.2107;
y=0;
plot(x,y,'oc5')
%=============================================
% construct leg2 2D workspace
h=[];
x=18.75:0.5:200;
for t=1:length(x)
  y=(0*x(:,t))-32.476;
  h=[h;x(:,t) y];
end
plot(h(:,1),h(:,2),'g')
%
k=[];
x=18.75:-0.5:-200;
for u=1:length(x)
  y=(1.73205*x(:,u))-64.9519;
  k=[k;x(:,u) y];
end
plot(k(:,1),k(:,2),'g')
%
% show leg2
L2=[];
x=18.75:0.5:89.1054;
for q=1:length(x)
  y=(-1.73025*x(:,q))+0;
  L2=[L2;x(:,q) y];
end
plot(L2(:,1),L2(:,2),'b')
%
% show foot2
x=89.1054;
y=-154.335;
plot(x,y,'oc5')
%=============================================
% construct leg5 2D workspace
l=[];
x=-18.75:0.5:200;
for v=1:length(x)
  y=(1.73205*x(:,v))+64.9519;
  l=[l;x(:,v) y];
end
plot(l(:,1),l(:,2),'g')
%
m=[];
x=-18.75:-0.5:-200;
for z=1:length(x)
  y=(0*x(:,z))+32.476;
  m=[m;x(:,z) y];
end
plot(m(:,1),m(:,2),'g')
%
% show leg5
L5=[];
x=-18.75:-0.5:-89.1054;
for q=1:length(x)
  y=(-1.73025*x(:,q))+0;
```

```
  L5=[L5;x(:,q) y];
end
plot(L5(:,1),L5(:,2),'b')
%
% show foot5
x=-89.1054;
y=154.335;
plot(x,y,'oc5')
%=================================
% construct leg3 2D workspace
l=[];
x=-18.75:0.5:200;
for v=1:length(x)
  y=(-1.73205*x(:,v))-64.9519;
  l=[l;x(:,v) y];
end
plot(l(:,1),l(:,2),'g')
%
m=[];
x=-18.75:-0.5:-200;
for z=1:length(x)
  y=(0*x(:,z))-32.476;
  m=[m;x(:,z) y];
end
plot(m(:,1),m(:,2),'g')
%
% show leg3
L3=[];
x=-18.75:-0.5:-89.1054;
for q=1:length(x)
  y=(1.73025*x(:,q))+0;
  L3=[L3;x(:,q) y];
end
plot(L3(:,1),L3(:,2),'b')
%
% show foot3
x=-89.1054;
y=-154.335;
plot(x,y,'oc5')
%=================================
% construct leg6 2D workspace
l=[];
x=18.75:-0.5:-200;
for v=1:length(x)
  y=(-1.73205*x(:,v))+64.9519;
  l=[l;x(:,v) y];
end
plot(l(:,1),l(:,2),'g')
%
m=[];
x=18.75:0.5:200;
for z=1:length(x)
  y=(0*x(:,z))+32.476;
  m=[m;x(:,z) y];
end
plot(m(:,1),m(:,2),'g')
%
% show leg6
L6=[];
x=18.75:0.5:89.1054;
for q=1:length(x)
  y=(1.73025*x(:,q))+0;
  L6=[L6;x(:,q) y];
end
plot(L6(:,1),L6(:,2),'b')
%
% show foot6
```

```
x=89.1054;
y=154.335;
plot(x,y,'oc5')
%=============================================
% label plot
title('AquaRobot Unconstrained Leg Workspace')
xlabel('X coordinate, cm')
ylabel('Y coordinate, cm')
gtext('Leg One')
meta work
```

APPENDIX C: "C++" PROGRAM LISTING

This appendix contains the real-time gait planning code written in the ANSI C++ language using the ATT version 3.1 compiler. These code modules are ready for implementation in the *AquaRobot* simulator (Silicon Graphics Personal Iris). These code modules, less the graphics portions, may also be used in the *AquaRobot* control computer (NEC 486-based Personal Computer) with slight modifications because of compiler differences.

```
// ********************************************************************
// FILENAME: AbBody.C
// PURPOSE: Implementation of the AquarobotBody class
// CONTAINS: initializes the body form
// ********************************************************************
#include "AbBody.H"

AquarobotBody::AquarobotBody()
{
 body_list = new matrix(7,4,0.0); // each row is a body point (x,y,z)
                  // the first (0 row) is the body's
                  // physical center, the rest are the six
                  // points of the body
 H_matrix = new matrix(4,4,0.0); // the body's H-matrix

 // the body design is constructed
 body_list->val(0,0) = 0.; body_list->val(0,1) = 0.;
 body_list->val(0,2) = 0.; body_list->val(0,3) = 1.;
 body_list->val(1,0) = 37.5; body_list->val(1,1) = 0.;
 body_list->val(1,2) = 0.; body_list->val(1,3) = 1.;
 body_list->val(2,0) = 18.75; body_list->val(2,1) = 32.48;
 body_list->val(2,2) = 0.; body_list->val(2,3) = 1.;
 body_list->val(3,0) = -18.75; body_list->val(3,1) = 32.48;
 body_list->val(3,2) = 0.; body_list->val(3,3) = 1.;
 body_list->val(4,0) = -37.5; body_list->val(4,1) = 0.;
 body_list->val(4,2) = 0.; body_list->val(4,3) = 1.;
 body_list->val(5,0) = -18.75; body_list->val(5,1) = -32.48;
 body_list->val(5,2) = 0.; body_list->val(5,3) = 1.;
 body_list->val(6,0) = 18.75; body_list->val(6,1) = -32.48;
 body_list->val(6,2) = 0.; body_list->val(6,3) = 1.;

 // defines the initial location of the body using the
 // H-matrix the inputs to the function are:
 //            (azimuth, elevation, roll, x, y, z)
 // H_matrix->HomogeneousTransform(0.,0.,0.,0., -54.1819);
 H_matrix->HomogeneousTransform(0.,0.,0.,0., -70.7107);

 // moves the body to the initial location desired
 body_list->TransformBodyList(*H_matrix,*body_list);

}

void AquarobotBody::MoveIncremental(double delaz, double delel,
     double delrol, double delx,double dely, double delz)
{
 double az, el, ro, x, y, z;
 az = azimuth + delaz;
 el = elevation + delel;
 ro = roll + delrol;
 x = body_list->val(0,0) + delx;
 y = body_list->val(0,1) + dely;
 z = body_list->val(0,2) + delz;

 // changes only are used since body_list is at current position
 H_matrix->HomogeneousTransform(delaz, delel, delrol, delx, dely, delz);
 body_list->TransformBodyList(*H_matrix,*body_list);

 // puts all info in H_matrix
 H_matrix->HomogeneousTransform(az,el,ro,x,y,z);

}
// ********************************************************************
// FILENAME: AbBody.H
// PURPOSE: Declaration of AquarobotBody class
//       Subclass of RigidBody class
// ********************************************************************
```

```
#ifndef H_AQUAROBOTBODY
#define H_AQUAROBOTBODY

#include <stdio.h>
#include "Link1.H"
#include "AbRigid.H"
#include "Link.H"
#include "MatrixMy.H"


class AquarobotBody : public RigidBody
{
 public:
  matrix *body_list; // defines the size of the body using coordinates

  matrix *H_matrix;  // defines the location for body
                // using azimuth, elevation, roll, x, y, z
  double azimuth, elevation, roll;

  AquarobotBody();   // constructor
  void MoveIncremental(double,double, double, double, double, double);

};

#endif

// ************************************************************************
// FILENAME: AbLeg.C
// PURPOSE: Implementation of AquaLeg class
// CONTAINS: AquaLeg()
//         Initialize(AquaLeg&, AquarobotBody&)
//         TakePicture(Camera&, AquaLeg&)
//         MoveIncremental (Aqualeg&, delta1,delta2,delta3)
// ************************************************************************
#include "AbLeg.H"

// ***************************************************************
// FUNCTION: ~AquaLeg()
// PURPOSE: destructor of AquaLeg class
// ***************************************************************
AquaLeg::~AquaLeg()
{
  delete link0;
  delete link1;
  delete link2;
  delete link3;
}

// ***************************************************************
// FUNCTION: AquaLeg
// PURPOSE:  constructor of AquaLeg class
// RETURNS:  AquaLeg class with values
// ***************************************************************
AquaLeg::AquaLeg(AquarobotBody &body, double angle)
{
  motion_complete_flag = 1;
  SetLegAttachmentAngle(angle);
  link0 = new Link0;
  link1 = new Link1;
  link2 = new Link2;
  link3 = new Link3;

  // initial link values initialized
  matrix temp;
  temp.UpdateTMatrix(GetLegAttachmentAngle(),0.,0.,0.);
  temp = *body.H_matrix * temp;
  link0->RotateLink(&temp ,link0->GetInboardJointAngle());
```

```
 link1->RotateLink(link0->H_matrix, link1->GetInboardJointAngle());
 link2->RotateLink(link1->H_matrix,link2->GetInboardJointAngle());
 link3->RotateLink(link2->H_matrix,link3->GetInboardJointAngle());
}

// ***************************************************************
// FUNCTION: MoveIncremental
// PURPOSE:  calculate the new link values as a leg rotates
// RETURNS:  rotated link's new values
// ***************************************************************
void AquaLeg::MoveIncremental(AquarobotBody &body  double delta1,
                                                        double delta2, double
delta3)
{
 double b;

 // set all limit flags to zero
 link1->SetMotionLimitFlag(0);
 link2->SetMotionLimitFlag(0);
 link3->SetMotionLimitFlag(0);

 // temp matrix adds in the T_matrix needed for the physical
 // attachment of the leg to the body
 matrix temp;
 temp.UpdateTMatrix(GetLegAttachmentAngle(),0.,0.,0.);
 temp = *body.H_matrix * temp;

 b = delta1 + link0->GetInboardJointAngle();
 link0->SetInboardJointAngle(b);
 link0->RotateLink(&temp,link0->GetInboardJointAngle());

 b = delta2 + link1->GetInboardJointAngle();
 link1->SetInboardJointAngle(b);
 link1->RotateLink(link0->H_matrix,link1->GetInboardJointAngle());

 b = delta3 + this->link2->GetInboardJointAngle();
 link2->SetInboardJointAngle(b);
 link2->Rotate(link1->H_matrix,link2->GetInboardJointAngle());

 b = delta3 + this->link3->GetInboardJointAngle();
 link3->SetInboardJointAngle(b);
 link3->RotateLink(link2->H_matrix,link3->GetInboardJointAngle());

 // the motion_complete_flag is set to 1 if the
 // motion_limit_flags on all legs are not set
 SetMotionCompleteFlag(!(this->link1->GetMotionLimitFlag() ||
                                                        this->link2->GetMotionLimitFlag() ||
                                                        this->link3->GetMotionLimitFlag()));


 // prints the status of the requested motion and prints which
 // link's motion_limit_flag was set (if any).
 if (GetMotionCompleteFlag() == 0) {
            printf("Motion Not Completed\n");
            if (link1->GetMotionLimitFlag() == 1)
                        printf("link 1 limit exceeded\n");
            if (link2->GetMotionLimitFlag() == 1)
                        printf("link 2 limit exceeded\n");
            if (link3->GetMotionLimitFlag() == 1)
                        printf("link 3 limit exceeded\n");
 }
 // else printf("Motion completed\n");

}

// ***************************************************************
// FILENAME: AbLeg.H
// PURPOSE: Declarations for AquaLeg class
```

```
//
// COMMENTS: Definition of AquaLeg class and functions that
//     apply to this class
// ****************************************************************

#ifndef H_AQUALEG
#define H_AQUALEG

#include "AbBody.H"
#include "Link.H"
#include "Link0.H"
#include "Link1.H"
#include "Link2.H"
#include "Link3.H"

class AquaLeg
{
 public:

 Link0 *link0;  // a Link0 class is instantiated
 Link1 *link1;  // a Link1 class is instantiated
 Link2 *link2;  // a Link2 class is instantiated
 Link3 *link3;  // a Link3 class is instantiated
 int motion_complete_flag; // the flag is set to 1 if the motion
                  // was completed without reaching any
                  // link limits.
 int leg_support_flag;    // the flag is set to 1 if the leg
                  // is on the floor (z = 0).
 double leg_attachment_angle; // the angle off of leg one where the
                    // leg is attached to the body

 double previous_foot_x_coord; // saves last foot position
 double previous_foot_y_coord;
 double previous_foot_z_coord;

 AquaLeg(AquarobotBody &, double); // constructor and initializer
 ~AquaLeg();            // destructor
 double GetLegAttachmentAngle() { return leg_attachment_angle;}
 int GetMotionCompleteFlag() { return motion_complete_flag;}
 void SetLegAttachmentAngle(double angle) {leg_attachment_angle = angle;}
 void SetMotionCompleteFlag(int flag) {motion_complete_flag = flag;}
 int GetLegSupportFlag() { return leg_support_flag;}
 void SetLegSupportFlag(int flag) {leg_support_flag = flag;}
 double GetPreviousFootXCoord() { return previous_foot_x_coord;}
 void SetPreviousFootXCoord(double xcoord) {previous_foot_x_coord = xcoord;}
 double GetPreviousFootYCoord() { return previous_foot_y_coord;}
 void SetPreviousFootYCoord(double ycoord) {previous_foot_y_coord = ycoord;}
 double GetPreviousFootZCoord() { return previous_foot_z_coord;}
 void SetPreviousFootZCoord(double zcoord) {previous_foot_z_coord = zcoord;}

 void MoveIncremental(AquarobotBody &,double delta1,double delta2,
                                                          double delta3);

};
#endif

// ********************************************************
// FILENAME: AbRigid.C
// PURPOSE: Implementation of class RigidBody
// CONTAINS:
// ********************************************************

#include "AbRigid.H"

// ********************************************************
// FUNCTION: RigidBody()
//*********************************************************
```

```
RigidBody::RigidBody()
{
 node_list = new matrix(8,4,0.0);
 H_matrix = new matrix(4,4,0.0);
};


// ***********************************************************************
// FUNCTION: UpdateVelocity
// PURPOSE:
// COMMENTS:
// ***********************************************************************
void RigidBody::UpdateVelocity(double delta_t)
{
 u = u + (delta_t * udot);
 v = v + (delta_t * vdot);
 w = w + (delta_t * wdot);
 p = p + (delta_t * pdot);
 q = q + (delta_t * qdot);
 r = r + (delta_t * rdot);
}

// ***********************************************************
// FILENAME: AbRigid.H
// PURPOSE:
//
// COMMENTS:
// ***********************************************************

#ifndef H_RIGIDBODY
#define H_RIGIDBODY

#include <time.h>

// #include "Link.H"
#include "MatrixMy.H"

const double gravity = 32.2185;


class RigidBody
{
private:

 // double location;
 double x, y, z;
 // velocity[6];
 double u, v, w, p, q, r;
 // acceleration[6];
 double udot, vdot, wdot, pdot, qdot, rdot;
 // forces_and_torques[6];
 double Fx, Fy, Fz, L, M, N;
 // moments_of_inertia[3];
 double Ix, Iy, Iz;
 double mass;
 long current_time;

public:
 matrix *node_list;
 matrix *H_matrix;
 RigidBody();
 ~RigidBody(){};     //more needed?
 void TransformNodeList();

 /*
 // Use these for dynamics::::::::
 matrix& TranslateAndEulerAngleTransform (double x, double y, double z,
```

```
            double azimuth,
            double elevation,
            double roll);
double GetDeltaT ();
void StartTimer ();
void UpdateRigidBody ();
*/

void UpdateAcceleration ();
void UpdateVelocity (double delta_t);

/*
void UpdateHMatrix (double delta_t);
void UpdatePosition ();
void GetNodePolygonList ();
*/

void SetU(double a) {u = a;}
void SetV(double a) {v = a;}
void SetW(double a) {w = a;}
void SetP(double a) {p = a;}
void SetQ(double a) {q = a;}
void SetR(double a) {r = a;}

double GetU() {return u;}
double GetV() {return v;}
double GetW() {return w;}
double GetP() {return p;}
double GetQ() {return q;}
double GetR() {return r;}

void SetUdot (double a) {udot = a;}
void SetVdot (double a) {vdot = a;}
void SetWdot (double a) {wdot = a;}
void SetPdot (double a) {pdot = a;}
void SetQdot (double a) {qdot = a;}
void SetRdot (double a) {rdot = a;}

double GetUdot() {return udot;}
double GetVdot() {return vdot;}
double GetWdot() {return wdot;}
double GetPdot() {return pdot;}
double GetQdot() {return qdot;}
double GetRdot() {return rdot;}

void SetFx(double a) {Fx = a;}
void SetFy(double a) {Fy = a;}
void SetFz(double a) {Fz = a;}
void SetL(double a) {L = a;}
void SetM(double a) {M = a;}
void SetN(double a) {N = a;}

double GetFx() {return Fx;}
double GetFy() {return Fy;}
double GetFz() {return Fz;}
double GetL() {return L;}
double GetM() {return M;}
double GetN() {return N;}

void SetIx(double a) {Ix = a;}
void SetIy(double a) {Iy = a;}
void SetIz(double a) {Iz = a;}

double GetIx() {return Ix;}
double GetIy() {return Iy;}
double GetIz() {return Iz;}
```

```
};

#endif

// ************************NPS*and*PHRI*********************************
//TITLE:   arcint.C
//INPUT:   (x,y)body coordinates of a selected foot; direction; body center;
//           radius of workspace; CCW arc segment endpoints; CW arc segment
//           endpoints; foot number selected
//OUTPUT:  intercepts of arc segment; distance to the intercepts from the
//           current foot position; flag to indicate whether an intercept
//           was found
//FUNCTION: This function determines whether there is an intercept of the
//           desired direction of the foot with an arc segment defining a part
//           of the workspace.
// ***********************AQUAROBOT*************************************

#include <stdio.h>
#include <math.h>

#define PI 3.14159265359

int arcint(double foot[], double direction, double cenbod[], double radius, double neg_limit[], double pos_limit[], int foot_number,
        double intercepts[], double &distance)
{
 // initialization
 double a, b, c, d, x_intercept, y_intercept;
 double perp_distance, length, x_perp, y_perp;
 double rad_squared, perp_squared;
 double radius_test, in_direction, out_direction, footrad;
 double arc_foot, neg_arc_limit, pos_arc_limit, limit = 215.0;
 int arcsegment_flag = 0;

 // determine if foot falls on the arc segment
 footrad = sqrt( (foot[0]*foot[0])+(foot[1]*foot[1]) );
 if (radius == footrad){
            arcsegment_flag = 1;
            intercepts[0] = foot[0];
            intercepts[1] = foot[1];
            distance = limit;
 }

if (arcsegment_flag == 0){
 perp_distance = ((cenbod[4] - foot[1])*cos(direction))
          - ((cenbod[3] - foot[0])*sin(direction));

 if (perp_distance <= radius){
  rad_squared = radius*radius;
  perp_squared = perp_distance*perp_distance;
  if (perp_distance < 0){
   length = sqrt (-(rad_squared - perp_squared));
            }
  else {
   length = sqrt(rad_squared - perp_squared);
            }

  // find (x,y) at intersection of perpindicular distance and length
  x_perp = cenbod[3] + (perp_distance*sin(direction));
  y_perp = cenbod[4] - (perp_distance*cos(direction));

  // test if foot is inside or outside radius
  a = (foot[1] - cenbod[4])*(foot[1] - cenbod[4]);
  b = (foot[0] - cenbod[3])*(foot[0] - cenbod[3]);
  radius_test = sqrt(a + b);

  if (radius_test > radius){
   // find (x,y) at intersection of ray and arc if foot is outside radius
```

```
  x_intercept = x_perp - (length*cos(direction));
  y_intercept = y_perp - (length*sin(direction));
  intercepts[0] = x_intercept;
  intercepts[1] = y_intercept;
 }
 else {
  // find (x,y) intersection of ray and arc if foot is inside radius
  x_intercept = x_perp + (length*cos(direction));
  y_intercept = y_perp + (length*sin(direction));
  intercepts[0] = x_intercept;
  intercepts[1] = y_intercept;
 } // radius test ends here

 // test if intersection is in desired direction
 in_direction = foot[0]*cos(direction) + foot[1]*sin(direction);
 out_direction = x_intercept*cos(direction) + y_intercept*sin(direction);

 if (in_direction <= out_direction){
  c = (y_intercept - foot[1])*(y_intercept - foot[1]);
  d = (x_intercept - foot[0])*(x_intercept - foot[0]);
  distance = sqrt(c + d);

  // test if intersection is within arc segment
  arc_foot = atan2(y_intercept - cenbod[4], x_intercept - cenbod[3]);
  neg_arc_limit = atan2(neg_limit[1] - cenbod[4], neg_limit[0] - cenbod[3]);
  pos_arc_limit = atan2(pos_limit[1] - cenbod[4], pos_limit[0] - cenbod[3]);

  // tests to ensure proper sections of arcs are used
  if (foot_number == 1){
          if (arc_foot >= neg_arc_limit && arc_foot <= pos_arc_limit)
            arcsegment_flag = 1;
  }
  else {
          if (direction > 0){
            if (neg_arc_limit < 0)
              neg_arc_limit = neg_arc_limit + (2*PI);
            if (pos_arc_limit < 0)
              pos_arc_limit = pos_arc_limit + (2*PI);
            if (arc_foot < 0)
              arc_foot = arc_foot + (2*PI);
            if (arc_foot >= neg_arc_limit && arc_foot <= pos_arc_limit)
              arcsegment_flag = 1;
          }
          else {  // direction < 0
            if (neg_arc_limit > 0)
              neg_arc_limit = neg_arc_limit - (2*PI);
                    if (pos_arc_limit > 0)
                      pos_arc_limit = pos_arc_limit - (2*PI);
                    if (arc_foot > 0)
                      arc_foot = arc_foot - (2*PI);
                    if (arc_foot >= neg_arc_limit && arc_foot <= pos_arc_limit)
                      arcsegment_flag = 1;
    } // direction test ends here
   } // foot number test ends here
  } // intersection test ends here
 } // perpindicular test ends here
}

 if (arcsegment_flag != 1){
  arcsegment_flag = 0;
            distance = limit;
  intercepts[0] = x_intercept;
  intercepts[1] = y_intercept;

 } // nullifying parameters ends here

 return (arcsegment_flag);
```

```
} // arcint function ends here

/*****************************************************************************
FILENAME: arconst.H
PURPOSE: Header file. This file represents Aquarobot's constant parameters
COMMENT:
*****************************************************************************/
#ifndef _ARCONST_H
#define _ARCONST_H

#define PI 3.14159265358979311599796346854418516159057617187500O

#define HPI (PI/2.0)
#define DR (PI/180.0)

#define ROOT3 1.732050807568877193176604123436845839023590087890625
#define EQU_TRIANGLE 3.0/2.0

#define LEG 6
#define LEG1 0
#define LEG2 1
#define LEG3 2
#define LEG4 3
#define LEG5 4
#define LEG6 5

#define CB 0
#define HIP 1
#define KNEE1 2
#define KNEE2 3
#define FOOT 4

#define VJOINT 4
#define JOINT 5
#define JOINT0 0
#define JOINT1 1
#define JOINT2 2
#define JOINT3 3
#define JOINT4 4

#define LINK0 37.5
#define LINK1 20.0
#define LINK2 50.0
#define LINK3 100.0

#define LINK02 (37.5*37.5)
#define LINK12 (20.0*20.0)
#define LINK22 (50.0*50.0)
#define LINK32 (100.0*100.0)

#define MAXMIN 2

#define TwoDim 2
#define ThreeDim 3
#define XY 2
#define XYZ 3
#define EULER 3
#define XW 0
#define YW 1
#define ZW 2
#define XB 0
#define YB 1
#define ZB 2
#define X 0
#define Y 1
#define Z 2
```

```
#define ResetThete1 (0.0)
#define ResetThete2 (66.4)
#define ResetThete3 (-156.4)

#define StartTheta0Leg1 0.0
#define StartTheta0Leg2 60.0
#define StartTheta0Leg3 120.0
#define StartTheta0Leg4 180.0
#define StartTheta0Leg5 240.0
#define StartTheta0Leg6 300.0

#define StartTheta1 (0.0)
#define StartTheta2 (35.86)
#define StartTheta3 (-125.86)

#define Thete1NegLim (-60.0)
#define Thete2NegLim (-106.6)
#define Thete3NegLim (-156.4)

#define Thete1PosLim (60.0)
#define Thete2PosLim (73.4)
#define Thete3PosLim (23.6)

#define StartX 98.02
#define StartY 0.0
#define StartZ 0.0

#define TripodSize 98.02
#define StartTripodSize 98.02
#define DefaultSTRIDE 120.0
#define DefaultFOOTheight 15.0

/*
#define HighHight   120.0
#define MidiumHight 100.0
#define LowHight    80.0
*/

#define InitialElpsRatio 2.0

#define J1LimN (-60.0*DR)
#define J1LimP ( 60.0*DR)
#define J2LimN (-106.6*DR)
#define J2LimP ( 73.4*DR)
#define J3LimN (-156.4*DR)
#define J3LimP ( 23.6*DR)

//#define ORIENTATION 3
#define AZIMUTH 0
#define ELEVATION 1
#define ROLL 2

#define YAW 0
#define PITCH 1
#define ROLL 2

/*
#define AZIMUTH 3
#define ELEVATION 4
#define ROLL 5

#define YAW 3
#define PITCH 4
#define ROLL 5
*/
```

```
#define FINE 50.0
#define DSMAX 5.0
#define DSinit 1.0
//#define SCONST (DSMAX/(90.0*90.0))

#define TP0 0
#define BODY0 1
#define TP1 2
#define BODY1 3

#endif
```

```
#ifndef _ARFUNC_C
#define _ARFUNC_C

// ****************************************************************************
// FILENAME: arfunc.C
// PURPOSE: Basic Mathematical Functions for Aquarobot Simulation.
// ****************************************************************************

#include <stdio.h>
#include <math.h>
#include "arconst.H"
#include "Kinematics.H"

// ****************************************************************************
// TITLE:   min()
// FUNCTION: minimum operation
// INPUT:
// OUTPUT:
// RETURN:
// COMMENT:
// DATE:
// ****************************************************************************
double min(double x, double y)
{
 if( x < y )
   return x;
 else
   return y;
}

// ****************************************************************************
// TITLE:   max()
// FUNCTION: maximum operation
// ****************************************************************************
double max(double x, double y)
{
 if( x > y )
   return x;
 else
   return y;
}

// ****************************************************************************
// TITLE:   ellipse()
// FUNCTION:  Calculates incremental foot point along an elliptical path
//          generated between the last step and the next step.
// INPUT:
// OUTPUT:
// CALLED BY: tripod0phase(), tripod1phase(), bodyphase().
// COMMENT:   This function is for continuous motion.
//          This function divides ellipse angle into FINE.
*****************************************************************************
void ellipse(WalkParameter &wp,  // walking parameters
                                    double footprint[], //
                                    double footpoint[]) //
{
 double theta;     // ellipse angle
 double d_theta;   // change in ellipse angle
//  double d_segment;  // ellipse segment speed
 double a, b;      // x_diameter, y_diameter
 double x, y, z;   // ellipse coordinates


 a = wp.stride / 2.0;
 b = wp.footheight;

 /* Coordinates Transformation(WORLD->ELLIPSE) */
```

```
x =  cos(wp.direction*DR)*(footpoint[XW] - footprint[XW])
  + sin(wp.direction*DR)*(footpoint[YW] - footprint[YW]);
y =- (footpoint[ZW] - footprint[ZW]);
z =- sin(wp.direction*DR)*(footpoint[XW] - footprint[XW])
          + cos(wp.direction*DR)*(footpoint[YW] - footprint[YW]);

/* to calculate current ellipse angle */
theta = atan2((y/b), (x-a)/a);

d_theta = PI/FINE;

/* decrement ellipse angle */
theta -= d_theta;

/* to calculate next foot point */
x = a*(1.0 + cos(theta));
y = b*sin(theta);
z = 0.0;

if (theta <= 0.0) {
          theta = 180.0;
}

/* to calculate foot position on World coordinates */
footpoint[XW] = x*cos(wp.direction*DR) - z*sin(wp.direction) + footprint[XW];
footpoint[YW] = x*sin(wp.direction*DR) + z*cos(wp.direction) + footprint[YW];
footpoint[ZW] =                          - y + footprint[ZW];

}

// ***************************************************************************
// TITLE:   kinematics()
// FUNCTION: Computes Kinematics for Aquarobot Type II.
// INPUT:   Joint Angles(theta[CB->FOOT] Unit:degree).
// OUTPUT:  Joint Positions related to BODY coordinates(Unit:cm).
// CALLED BY:inv_kinematics(), tripod0phase(), tripod1phase(), bodyphase()
// ***************************************************************************
void kinematics(double theta[], // Joint Angle
                                        double hip[],   // HIP Joint Position(BODY)
                                        double knee1[], // KNEE1 Joint Position(BODY)
                                        double knee2[], // KNEE2 Joint Position(BODY)
                                        double foot[]) // FOOT Joint Position(BODY)
{
 double c0, s0, c1, s1, c2, s2, c3, s3;
 double c01, s01, c23, s23;
 int i;

 for(i=0; i<4; i++){
  theta[i] = theta[i]*DR;
 }

 c0  = cos(theta[0]);  s0  = sin(theta[0]);
 c1  = cos(theta[1]);  s1  = sin(theta[1]);
 c2  = cos(theta[2]);  s2  = sin(theta[2]);
 c3  = cos(theta[3]);  s3  = sin(theta[3]);
 c01 = cos(theta[0]+theta[1]);  s01 = sin(theta[0]+theta[1]);
 c23 = cos(theta[2]+theta[3]);  s23 = sin(theta[2]+theta[3]);

 hip[XB] = LINK0*c0; // HIP Position
 hip[YB] = LINK0*s0;
 hip[ZB] = 0.0;

 knee1[XB] = hip[XB] + LINK1*c01; // KNEE1 Position
 knee1[YB] = hip[YB] + LINK1*s01;
 knee1[ZB] = 0.0;

 knee2[XB] = knee1[XB] + LINK2*c01*c2; // KNEE2 Position
```

```
 knee2[YB] = knee1[YB] + LINK2*s01*c2;
 knee2[ZB] =-LINK2*s2;

 foot[XB] = knee2[XB] + LINK3*c01*c23; // FOOT Position
 foot[YB] = knee2[YB] + LINK3*s01*c23;
 foot[ZB] = knee2[ZB] - LINK3*s23;

 for(i=0; i<4; i++){
  theta[i] = theta[i]/(DR);
 }
}


// ***************************************************************************
// TITLE:   inv_kinematics()
// FUNCTION: Computes Inverse Kinematics for Aquarobot Type II.
// INPUT:   FOOT position in World Coordinates(foot[XYZ]).
// OUTPUT:  Joint Angles(Unit:degree).
// CALLED BY:tripod0phase(), tripod1phase(), bodyphase()
// CALLS:   kinematics()
// ***************************************************************************
void inv_kinematics(double foot[], // FOOT Position(BODY)
                                                    double theta[])// Joint Angle
{
 double px, py, pz;
 double px2, py2, pz2;      /* px^2, py^2, pz^2 */
 double x0, y0, z0;         /* CB coordinates origin */
 double c0, s0, c1, s1;     /* sin(theta_i),       cos(theta_j) */
 double c3, s3;             /* sin(theta_i),       cos(theta_j) */
 double c01, s01;           /* sin(theta_i+theta_j), cos(theta_i+theta_j) */
 double b1, b2, b3;         /* length bi*/
 double b12, b22, b32;      /* bi^2 */
 double beta;               /* angle beta */
 double cpsi, spsi;         /* cos(psi), sin(psi) */
 double theta2_p, theta2_n;
 double theta3_p, theta3_n;
 double flag;

 double th[4];              /* joint angle */
 double hp[3], k1[3], k2[3], ft[3];/* joint position */
 int i;

 px = foot[0]; px2 = px*px;
 py = foot[1]; py2 = py*py;
 pz = foot[2]; pz2 = pz*pz;

 theta[0] = theta[0]*DR;
 c0  = cos(theta[0]);
 s0  = sin(theta[0]);
 b12 = (px-LINK0*c0)*(px-LINK0*c0) + (py-LINK0*s0)*(py-LINK0*s0);
 b1  = sqrt(b12);
 b2  = b1-LINK1;
 b22 = b2*b2;
 b32 = b22 + pz2;
 b3  = sqrt(b32);

 /* theta1 */
 c1 = ( px2+py2-LINK02-b12 )/( 2.0*LINK0*b1 );
 c1 = min(1.0, c1);
 s1 = sqrt(1.0-c1*c1);

 /* position inverse transformation from CB to J0 coordinates */
 x0 = px*c0 + py*s0;
 y0 =-px*s0 + py*c0;
 z0 = pz;

 if( x0 <= 0.0 ){
```

```
  printf("##### XB range error #####\n");
 }
 if( y0 >= 0.0 ){
  theta[1] = atan2( s1, c1);
 }
 else{
  theta[1] = atan2(-s1, c1);
 }

 if( (theta[1] < J1LimN) && (theta[1] > J1LimP) ){
  printf("##### NO SOLUTION FOR THETA1 #####\n");
 }

 c01 = cos(theta[0]+theta[1]);
 s01 = sin(theta[0]+theta[1]);
 theta[0] = theta[0]/(DR);
 theta[1] = theta[1]/(DR);

 /* theta2 */
 beta = atan2( pz, b2 );
 cpsi = min(1.0, ( LINK22+b32-LINK32 )/( 2.0*LINK2*b3 ));
 spsi = sqrt(1.0-cpsi*cpsi);

 theta2_p = ( atan2( spsi, cpsi) - beta)/(DR);
 theta2_n = ( atan2(-spsi, cpsi) - beta)/(DR);

 /* theta3 */
 c3 = min(1.0, ( b32-LINK22-LINK32 )/( 2.0*LINK2*LINK3 ));
 s3 = sqrt(1.0-c3*c3);
 theta3_p = atan2( s3, c3)/(DR);
 theta3_n = atan2(-s3, c3)/(DR);

 /* selection of proper combination of theta2 and theta3 */
 th[0]=theta[0]; th[1]=theta[1];
 flag=0;
 for(i=0; i<4; i++){
  switch(i){
  case 0:
   th[2] = theta2_p;  th[3] = theta3_n;
   break;
  case 1:
   th[2] = theta2_n;  th[3] = theta3_n;
   break;
  case 2:
   th[2] = theta2_n;  th[3] = theta3_p;
   break;
  case 3:
   th[2] = theta2_p;  th[3] = theta3_p;
   break;
  }
  kinematics(th, hp, k1, k2, ft);
  if(  (fabs(px-ft[XB]) < 1.0e-3)
    && (fabs(py-ft[YB]) < 1.0e-3)
    && (fabs(pz-ft[ZB]) < 1.0e-3) ){
   theta[2] = th[2];
   theta[3] = th[3];
   flag=1;
   break;
  }
 }
 if(flag==0)
  printf("##### NO SOLUTION FOR THETA2 & THETA3 ##### \n");
}

// ****************************************************************************
// TITLE:   world_body()
// FUNCTION: Coordinate transformation from BODY to WORLD
```

```
// INPUTS:  euler angles(euler[psi,theta,phi]), Origin of BODY coordinates.
//                    Azimuth,Elevation,Roll
//                        Yaw,Pitch,Roll
// OUTPUTS: WORLD coordinates
// CALLED BY:tripod0phase(), tripod1phase(), bodyphase()
// ***************************************************************************
void world_body(double euler[],  // Euler angles
                double org[],    // Origin of BODY coordinates on WORLD
                double position[])// coordinates
{
 double c1, c2, c3, s1, s2, s3;
 double x, y, z;

 c1 = cos(euler[AZIMUTH]);  s1=sin(euler[AZIMUTH]);
 c2 = cos(euler[ELEVATION]); s2=sin(euler[ELEVATION]);
 c3 = cos(euler[ROLL]);     s3=sin(euler[ROLL]);

 x = position[XB];
 y = position[YB];
 z = position[ZB];

 position[XW] = c1*c2*x + (c1*s2*s3-s1*c3)*y + (c1*s2*c3+s1*s3)*z + org[X];
 position[YW] = s1*c2*x + (s1*s2*s3+c1*c3)*y + (s1*s2*c3-c1*s3)*z + org[Y];
 position[ZW] =  -s2*x +          c2*s3*y +          c2*c3*z + org[Z];
}

// ***************************************************************************
// TITLE:   body_world()
// FUNCTION: Coordinate transformation from WORLD to BODY
// INPUTS:  euler angles(euler[psi,theta,phi]), Origin of BODY coordinates.
//                    Azimuth,Elevation,Roll
//                        Yaw,Pitch,Roll
// OUTPUTS: BODY coordinates
// CALLED BY:tripod0phase(), tripod1phase(), bodyphase()
// ***************************************************************************
void body_world(double euler[],  // euler angle
                                          double org[],     // origin of BODY coordinates on WORLD
                                          double position[]) // coordinates
{
 double c1, c2, c3, s1, s2, s3; /* cos(), sin() */
 double x, y, z;            /* WORLD coordinates */

 c1=cos(euler[AZIMUTH]);  s1=sin(euler[AZIMUTH]);
 c2=cos(euler[ELEVATION]); s2=sin(euler[ELEVATION]);
 c3=cos(euler[ROLL]);     s3=sin(euler[ROLL]);

 x = position[XW];
 y = position[YW];
 z = position[ZW];

 position[XB] = c1*c2*(x-org[XW]) + s1*c2*(y-org[YW]) - s2*(z-org[ZW]);
 position[YB] = (c1*s2*s3-s1*c3)*(x-org[XW])
        +(s1*s2*s3+c1*c3)*(y-org[YW]) + c2*s3*(z-org[ZW]);
 position[ZB] = (c1*s2*c3+s1*s3)*(x-org[XW])
        +(s1*s2*c3-c1*s3)*(y-org[YW]) + c2*c3*(z-org[ZW]);
}

#endif

// ************************NPS*and*PHRI*********************************
//TITLE:   maxdistance_25cmfoot()
//INPUT:   (x,y)body coordinates of all six feet and body center; tripod
//              number (0 = tripod 0, 2 = tripod 1); direction
//OUTPUT:  maximum stride possible without exceeding workspace
//FUNCTION: This function finds the maximum stride possible for AquaRobot,
//              using a tripod gait, given an arbitrary direction as an input.
// ************************AQUAROBOT*********************************
```

```
double maxdistance_25cmfoot(Next_Motion &nm, WalkParameter &wp)
{

 int arcint(double*, double, double*, double, double*, double*, int, double*, double &);
 int segint(double*, double, double*, double*, double*, double*, double &);

 // initialization
 double inrad = 37.5;     // inner radius of workspace
 double outrad = 178.2197; // outer radius of workspace
 double maxdist, distance_a, distance_b, distance_c, limit = 36.51;
 double intercepts[2];
 double mdistance = 0.0;
 int intercept_flag = 0;
 int footnum, convert;
 double body_center[3];
 double euler[3];

 double afoot[3];
 double bfoot[3];
 double cfoot[3];
 double dfoot[3];
 double efoot[3];
 double ffoot[3];
 double cenbod[6];
 for (convert = 0; convert <= 2; convert++) {
            afoot[convert] = nm.foot_1_coord[convert];
            bfoot[convert] = nm.foot_2_coord[convert];
            cfoot[convert] = nm.foot_3_coord[convert];
            dfoot[convert] = nm.foot_4_coord[convert];
            efoot[convert] = nm.foot_5_coord[convert];
            ffoot[convert] = nm.foot_6_coord[convert];
 }
 for (convert = 0; convert <= 5; convert++) {
            cenbod[convert] = nm.body_center_coord[convert];
 }

 double dir = wp.direction * DR;
 int tripod = wp.phase;

 for(convert = 0; convert <= 2; convert++) {
            euler[convert] = nm.body_center_coord[convert];
            body_center[convert] = nm.body_center_coord[convert+3];
 }

 static double seg1pin[2] = {36.8686, 6.8524};
 // inside endpoint of CW segment #1
 static double seg1pout[2] = {160.2049, 78.0606};
 // outside endpoint of CW segment #1
 static double seg1nin[2] = {36.8686, -6.8524};
 // inside endpoint of CCW segment #1
 static double seg1nout[2] = {160.2049, -78.0606};
 // outside endpoint of CCW segment #1

 static double seg2pin[2] = {12.5, 35.3553};
 // inside endpoint of CW segment #2
 static double seg2pout[2] = {12.5, 177.7718};
 // outside endpoint of CW segment #2
 static double seg2nin[2] = {24.3686, 28.5030};
 // inside endpoint of CCW segment #2
 static double seg2nout[2] = {147.7049, 99.7112};
 // outside endpoint of CCW segment #2

 static double seg3pin[2] = {-24.3686, 28.5030};
 // inside endpoint of CW segment #3
 static double seg3pout[2] = {-147.7049, 99.7112};
 // outside endpoint of CW segment #3
```

```
static double seg3nin[2] = {-12.5, 35.3553};
// inside endpoint of CCW segment #3
static double seg3nout[2] = {-12.5, 177.7718};
// outside endpoint of CCW segment #3

static double seg4pin[2] = {-36.8686, -6.8524};
// inside endpoint of CW segment #4
static double seg4pout[2] = {-160.2049, -78.0606};
// outside endpoint of CW segment #4
static double seg4nin[2] = {-36.8686, 6.8524};
// inside endpoint of CCW segment #4
static double seg4nout[2] = {-160.2049, 78.0606};
// outside endpoint of CCW segment #4

static double seg5pin[2] = {-12.5, -35.3553};
// inside endpoint of CW segment #5
static double seg5pout[2] = {-12.5, -177.7718};
// outside endpoint of CW segment #5
static double seg5nin[2] = {-24.3686, -28.5030};
// inside endpoint of CCW segment #5
static double seg5nout[2] = {-147.7049, -99.7112};
// outside endpoint of CCW segment #5

static double seg6pin[2] = {24.3686, -28.5030};
// inside endpoint of CW segment #6
static double seg6pout[2] = {147.7049, -99.7112};
// outside endpoint of CW segment #6
static double seg6nin[2] = {12.5, -35.3553};
// inside endpoint of CCW segment #6
static double seg6nout[2] = {12.5, -177.7718};
// outside endpoint of CCW segment #6

// convert from WORLD to BODY coordinates
body_world(euler, body_center, afoot);
body_world(euler, body_center, bfoot);
body_world(euler, body_center, cfoot);
body_world(euler, body_center, dfoot);
body_world(euler, body_center, efoot);
body_world(euler, body_center, ffoot);
body_world(euler, body_center, &cenbod[3]);

if (tripod == 0){

 // test foot1 in leg 1 workspace
 footnum = 1; // select foot number one
 // test for intersection with outer workspace limit

 intercept_flag = arcint(afoot, dir, cenbod, outrad, seg1nout, seg1pout, footnum, intercepts, mdistance);
 if (intercept_flag == 1)
   distance_a = mdistance;

 // test for intersection with inner workspace limit
 intercept_flag = arcint(afoot, dir, cenbod, inrad, seg1nin, seg1pin, footnum, intercepts, mdistance);
 if (intercept_flag == 1)
   distance_a = mdistance;

 // test for intersection with CW workspace limit
 intercept_flag = segint(afoot, dir, seg1pin, seg1pout, intercepts, mdistance);
 if (intercept_flag == 1)
   distance_a = mdistance;

 // test for intersection with CCW workspace limit
 intercept_flag = segint(afoot, dir, seg1nin, seg1nout, intercepts, mdistance);
 if (intercept_flag == 1)
   distance_a = mdistance;

 // test foot3 in leg 3 workspace
```

```
  footnum = 3; // select foot number three
  // test for intersection with outer workspace limit
  intercept_flag = arcint(cfoot, dir, cenbod, outrad,
                                    seg3nout, seg3pout, footnum, intercepts, mdistance);
  if (intercept_flag == 1)
   distance_b = mdistance;

  // test for intersection with inner workspace limit
  intercept_flag = arcint(cfoot, dir, cenbod, inrad,
                                    seg3nin, seg3pin, footnum, intercepts, mdistance);
  if (intercept_flag == 1)
   distance_b = mdistance;

  // test for intersection with CW workspace limit
  intercept_flag = segint(cfoot, dir, seg3pin, seg3pout,
                                    intercepts, mdistance);
  if (intercept_flag == 1)
   distance_b = mdistance;

  // test for intersection with CCW workspace limit
  intercept_flag = segint(cfoot, dir, seg3nin, seg3nout,
                                    intercepts, mdistance);
  if (intercept_flag == 1)
   distance_b = mdistance;

  // test foot5 in leg 5 workspace
  footnum = 5; // select foot number five
  // test for intersection with outer workspace limit
  intercept_flag = arcint(efoot, dir, cenbod, outrad,
                                    seg5nout, seg5pout, footnum, intercepts, mdistance);
  if (intercept_flag == 1)
   distance_c = mdistance;

  // test for intersection with inner workspace limit
  intercept_flag = arcint(efoot, dir, cenbod, inrad,
                                    seg5nin, seg5pin, footnum, intercepts, mdistance);
  if (intercept_flag == 1)
   distance_c = mdistance;

  // test for intersection with CW workspace limit
  intercept_flag = segint(efoot, dir, seg5pin, seg5pout,
                                    intercepts, mdistance);
  if (intercept_flag == 1)
   distance_c = mdistance;

  // test for intersection with CCW workspace limit
  intercept_flag = segint(efoot, dir, seg5nin, seg5nout,
                                    intercepts, mdistance);
  if (intercept_flag == 1)
   distance_c = mdistance;

maxdist = min(min(distance_a, distance_b), distance_c);

 } // this ends tripod 0 testing

 else if (tripod == 2){

  // test foot2 in leg 2 workspace
  footnum = 2; // select foot number two
  // test for intersection with outer workspace limit
  intercept_flag = arcint(bfoot, dir, cenbod, outrad,
                                    seg2nout, seg2pout, footnum, intercepts, mdistance);
  if (intercept_flag == 1)
   distance_a = mdistance;

  // test for intersection with inner workspace limit
   intercept_flag = arcint(bfoot, dir, cenbod, inrad,
```

```
                                    seg2nin, seg2pin, footnum, intercepts, mdistance);
if (intercept_flag == 1)
 distance_a = mdistance;

// test for intersection with CW workspace limit
 intercept_flag = segint(bfoot, dir, seg2pin, seg2pout,
                                    intercepts, mdistance);
if (intercept_flag == 1)
 distance_a = mdistance;

// test for intersection with CCW workspace limit
 intercept_flag = segint(bfoot, dir, seg2nin, seg2nout,
                                    intercepts, mdistance);
if (intercept_flag == 1)
 distance_a = mdistance;

// test foot4 in leg 4 workspace
footnum = 4; // select foot number four
// test for intersection with outer workspace limit
intercept_flag = arcint(dfoot, dir, cenbod, outrad,
                                   seg4nout, seg4pout, footnum, intercepts, mdistance);
if (intercept_flag == 1)
 distance_b = mdistance;

// test for intersection with inner workspace limit
intercept_flag = arcint(dfoot, dir, cenbod, inrad,
                                   seg4nin, seg4pin, footnum, intercepts, mdistance);
if (intercept_flag == 1)
 distance_b = mdistance;

// test for intersection with CW workspace limit
intercept_flag = segint(dfoot, dir, seg4pin, seg4pout,
                                   intercepts, mdistance);
if (intercept_flag == 1)
 distance_b = mdistance;

// test for intersection with CCW workspace limit
 intercept_flag = segint(dfoot, dir, seg4nin, seg4nout,
                                    intercepts, mdistance);
if (intercept_flag == 1)
 distance_b = mdistance;

// test foot6 in leg 6 workspace
footnum = 6; // select foot number six
 // test for intersection with outer workspace limit
 intercept_flag = arcint(ffoot, dir, cenbod, outrad, seg6nout, seg6pout,
                                    footnum, intercepts, mdistance);
if (intercept_flag == 1)
 distance_c = mdistance;

// test for intersection with inner workspace limit
intercept_flag = arcint(ffoot, dir, cenbod, inrad, seg6nin, seg6pin,
                                   footnum, intercepts, mdistance);
if (intercept_flag == 1)
 distance_c = mdistance;

// test for intersection with CW workspace limit
intercept_flag = segint(ffoot, dir, seg6pin, seg6pout,
                                   intercepts, mdistance);
if (intercept_flag == 1)
 distance_c = mdistance;

// test for intersection with CCW workspace limit
intercept_flag = segint(ffoot, dir, seg6nin, seg6nout,
                                   intercepts, mdistance);
if (intercept_flag == 1)
 distance_c = mdistance;
```

```
maxdist = min(min(distance_a, distance_b), distance_c);

 } // this ends tripod 1 testing

 return (maxdist);

} // this ends function maxdistance_25cmfoot

// ********************NPS*and*PHRI***********************************
//TITLE:   maxdistance_45cmfoot()
//INPUT:   (x,y)body coordinates of all six feet and body center; tripod
//               number (0 = tripod 0, 2 = tripod 1); direction
//OUTPUT:  maximum stride possible without exceeding workspace
//FUNCTION: This function finds the maximum stride possible for AquaRobot,
//               using a tripod gait, given an arbitrary direction as an input.
// ************************AQUAROBOT***********************************

double maxdistance_45cmfoot(Next_Motion &nm, WalkParameter &wp)
{

 int arcint(double*, double, double*, double, double*, double*, int, double*, double &);
 int segint(double*, double, double*, double*, double*, double &);

 // initialization
 double inrad = 45.0;     // inner radius of workspace
 double outrad = 149.27; // outer radius of workspace
 double maxdist, distance_a, distance_b, distance_c;
 double intercepts[2];
 double distance = 0;
 int intercept_flag = 0;
 int footnum;

 double afoot[3];
 double bfoot[3];
 double cfoot[3];
 double dfoot[3];
 double efoot[3];
 double ffoot[3];
 double cenbod[6];
 afoot[3] = nm.foot_1_coord[3];
 bfoot[3] = nm.foot_2_coord[3];
 cfoot[3] = nm.foot_3_coord[3];
 dfoot[3] = nm.foot_4_coord[3];
 efoot[3] = nm.foot_5_coord[3];
 ffoot[3] = nm.foot_6_coord[3];
 cenbod[6] = nm.body_center_coord[6];
 double dir = wp.direction;
 int tripod = wp.phase;

 static double seg1pin[2] = {45.0, 0.0};
 // inside endpoint of CW segment #1
 static double seg1pout[2] = {139.0446, 54.2967};
 // outside endpoint of CW segment #1
 static double seg1nin[2] = {45.0, 0.0};
 // inside endpoint of CCW segment #1
 static double seg1nout[2] = {139.0446, -54.2967};
 // outside endpoint of CCW segment #1

 static double seg2pin[2] = {22.5, 38.9711};
 // inside endpoint of CW segment #2
 static double seg2pout[2] = {22.5, 147.5645};
 // outside endpoint of CW segment #2
 static double seg2nin[2] = {22.5, 38.9711};
 // inside endpoint of CCW segment #2
 static double seg2nout[2] = {116.5446, 93.2678};
 // outside endpoint of CCW segment #2
```

```
static double seg3pin[2] = {-22.5, 38.9711};
// inside endpoint of CW segment #3
static double seg3pout[2] = {-116.5446, 93.2678};
// outside endpoint of CW segment #3
static double seg3nin[2] = {-22.5, 38.9711};
// inside endpoint of CCW segment #3
static double seg3nout[2] = {-22.5, 147.5645};
// outside endpoint of CCW segment #3

static double seg4pin[2] = {-45.0, 0.0};
// inside endpoint of CW segment #4
static double seg4pout[2] = {-139.0446, -54.2967};
// outside endpoint of CW segment #4
static double seg4nin[2] = {-45.0, 0.0};
// inside endpoint of CCW segment #4
static double seg4nout[2] = {-139.0446, 54.2967};
// outside endpoint of CCW segment #4

static double seg5pin[2] = {-22.5, -38.9711};
// inside endpoint of CW segment #5
static double seg5pout[2] = {-22.5, -147.5645};
// outside endpoint of CW segment #5
static double seg5nin[2] = {-22.5, -38.9711};
// inside endpoint of CCW segment #5
static double seg5nout[2] = {-116.5446, -93.2678};
// outside endpoint of CCW segment #5

static double seg6pin[2] = {22.5, -38.9711};
// inside endpoint of CW segment #6
static double seg6pout[2] = {116.5446, -93.2678};
// outside endpoint of CW segment #6
static double seg6nin[2] = {22.5, -38.9711};
// inside endpoint of CCW segment #6
static double seg6nout[2] = {22.5, -147.5645};
// outside endpoint of CCW segment #6

if (tripod == 0){

 // test foot1 in leg 1 workspace
 footnum = 1; // select foot number one
 // test for intersection with outer workspace limit

 intercept_flag = arcint(afoot, dir, cenbod, outrad, seg1nout, seg1pout, footnum, intercepts, distance);
 if (intercept_flag == 1)
   distance_a = distance;

 // test for intersection with CW workspace limit
 intercept_flag = segint(afoot, dir, seg1pin, seg1pout, intercepts, distance);
 if (intercept_flag == 1)
   distance_a = distance;

 // test for intersection with CCW workspace limit
 intercept_flag = segint(afoot, dir, seg1nin, seg1nout, intercepts, distance);
 if (intercept_flag == 1)
   distance_a = distance;

 // test foot3 in leg 3 workspace
 footnum = 3; // select foot number three
 // test for intersection with outer workspace limit
 intercept_flag = arcint(cfoot, dir, cenbod, outrad,
                                    seg3nout, seg3pout, footnum, intercepts, distance);
 if (intercept_flag == 1)
   distance_b = distance;

  // test for intersection with CW workspace limit
 intercept_flag = segint(cfoot, dir, seg3pin, seg3pout,
```

```
                                        intercepts, distance);
if (intercept_flag == 1)
  distance_b = distance;

// test for intersection with CCW workspace limit
intercept_flag = segint(cfoot, dir, seg3nin, seg3nout,
                                        intercepts, distance);
if (intercept_flag == 1)
  distance_b = distance;

// test foot5 in leg 5 workspace
footnum = 5; // select foot number five
// test for intersection with outer workspace limit
intercept_flag = arcint(efoot, dir, cenbod, outrad,
                                        seg5nout, seg5pout, footnum, intercepts, distance);
if (intercept_flag == 1)
  distance_c = distance;

// test for intersection with CW workspace limit
intercept_flag = segint(efoot, dir, seg5pin, seg5pout,
                                        intercepts, distance);
if (intercept_flag == 1)
  distance_c = distance;

// test for intersection with CCW workspace limit
intercept_flag = segint(efoot, dir, seg5nin, seg5nout,
                                        intercepts, distance);
if (intercept_flag == 1)
  distance_c = distance;

} // this ends tripod 0 testing

else if (tripod == 2){

// test foot2 in leg 2 workspace
footnum = 2; // select foot number two
// test for intersection with outer workspace limit
intercept_flag = arcint(bfoot, dir, cenbod, outrad,
                                        seg2nout, seg2pout, footnum, intercepts, distance);
if (intercept_flag == 1)
  distance_a = distance;

// test for intersection with CW workspace limit
  intercept_flag = segint(bfoot, dir, seg2pin, seg2pout,
                                          intercepts, distance);
if (intercept_flag == 1)
  distance_a = distance;

// test for intersection with CCW workspace limit
  intercept_flag = segint(bfoot, dir, seg2nin, seg2nout,
                                          intercepts, distance);
if (intercept_flag == 1)
  distance_a = distance;

// test foot4 in leg 4 workspace
footnum = 4; // select foot number four
// test for intersection with outer workspace limit
intercept_flag = arcint(dfoot, dir, cenbod, outrad.
                                        seg4nout, seg4pout, footnum, intercepts, distance);
if (intercept_flag == 1)
  distance_b = distance;

// test for intersection with CW workspace limit
intercept_flag = segint(dfoot, dir, seg4pin, seg4pout,
                                        intercepts, distance);
if (intercept_flag == 1)
  distance_b = distance;
```

```
  // test for intersection with CCW workspace limit
    intercept_flag = segint(dfoot, dir, seg4nin, seg4nout,
                                          intercepts, distance);
  if (intercept_flag == 1)
    distance_b = distance;

  // test foot6 in leg 6 workspace
  footnum = 6; // select foot number six
    // test for intersection with outer workspace limit
    intercept_flag = arcint(ffoot, dir, cenbod, outrad, seg6nout, seg6pout,
                                          footnum, intercepts, distance);
  if (intercept_flag == 1)
    distance_c = distance;

  // test for intersection with CW workspace limit
  intercept_flag = segint(ffoot, dir, seg6pin, seg6pout,
                                          intercepts, distance);
  if (intercept_flag == 1)
    distance_c = distance;

  // test for intersection with CCW workspace limit
  intercept_flag = segint(ffoot, dir, seg6nin, seg6nout,
                                          intercepts, distance);
  if (intercept_flag == 1)
    distance_c = distance;

 } // this ends tripod 1 testing

 maxdist = min(min(distance_a, distance_b), distance_c);

 return (maxdist);

} // this ends function maxdistance_45cmfoot

// ************************NPS*and*PHRI************************************
//TITLE:   arcint()
//INPUT:   (x,y)body coordinates of a selected foot; direction; body center;
//           radius of workspace; CCW arc segment endpoints; CW arc segment
//           endpoints; foot number selected
//OUTPUT:  intercepts of arc segment; mdistance to the intercepts from the
//           current foot position; flag to indicate whether an intercept
//           was found
//FUNCTION: This function determines whether there is an intercept of the
//           desired direction of the foot with an arc segment defining a part
//           of the workspace.
// ************************AQUAROBOT************************************

int arcint(double foot[], double direction, double cenbod[], double radius, double neg_limit[], double pos_limit[], int foot_number,
      double intercepts[], double &mdistance)
{
 // initialization
 double a, b, c, d, x_intercept, y_intercept;
 double perp_distance, length, x_perp, y_perp;
 double rad_squared, perp_squared;
 double radius_test, in_direction, out_direction;
 double arc_foot, neg_arc_limit, pos_arc_limit;
 int arcsegment_flag = 0;

 perp_distance = ((cenbod[4] - foot[1])*cos(direction))
         - ((cenbod[3] - foot[0])*sin(direction));
 if (perp_distance <= radius){
  rad_squared = radius*radius;
  perp_squared = perp_distance*perp_distance;
  if (perp_distance < 0)
   length = sqrt (-(rad_squared - perp_squared));
  else
```

```
 length = sqrt(rad_squared - perp_squared);

// find (x,y) at intersection of perpindicular mdistance and length
x_perp = cenbod[3] + (perp_distance*sin(direction));
y_perp = cenbod[4] - (perp_distance*cos(direction));

// test if foot is inside or outside radius
a = (foot[1] - cenbod[4])*(foot[1] - cenbod[4]);
b = (foot[0] - cenbod[3])*(foot[0] - cenbod[3]);
radius_test = sqrt(a + b);

if (radius_test > radius){
 // find (x,y) at intersection of ray and arc if foot is outside radius
 x_intercept = x_perp - (length*cos(direction));
 y_intercept = y_perp - (length*sin(direction));
 intercepts[0] = x_intercept;
 intercepts[1] = y_intercept;
}
else {
 // find (x,y) intersection of ray and arc if foot is inside radius
 x_intercept = x_perp + (length*cos(direction));
 y_intercept = y_perp + (length*sin(direction));
 intercepts[0] = x_intercept;
 intercepts[1] = y_intercept;
} // radius test ends here

// test if intersection is in desired direction
in_direction = foot[0]*cos(direction) + foot[1]*sin(direction);
out_direction = x_intercept*cos(direction) + y_intercept*sin(direction);

if (in_direction <= out_direction){
 c = (y_intercept - foot[1])*(y_intercept - foot[1]);
 d = (x_intercept - foot[0])*(x_intercept - foot[0]);
 mdistance = sqrt(c + d);

 // test if intersection is within arc segment
 arc_foot = atan2(y_intercept - cenbod[4], x_intercept - cenbod[3]);
 neg_arc_limit = atan2(neg_limit[1] - cenbod[4], neg_limit[0] - cenbod[3]);
 pos_arc_limit = atan2(pos_limit[1] - cenbod[4], pos_limit[0] - cenbod[3]);

 // tests to ensure proper sections of arcs are used
 if (foot_number == 1){
        if (arc_foot >= neg_arc_limit && arc_foot <= pos_arc_limit)
          arcsegment_flag = 1;
 }
 else {
        if (direction > 0){
          if (neg_arc_limit < 0)
            neg_arc_limit = neg_arc_limit + (2*PI);
          if (pos_arc_limit < 0)
            pos_arc_limit = pos_arc_limit + (2*PI);
          if (arc_foot < 0)
            arc_foot = arc_foot + (2*PI);
          if (arc_foot >= neg_arc_limit && arc_foot <= pos_arc_limit)
            arcsegment_flag = 1;
        }
        else {  // direction < 0
          if (neg_arc_limit > 0)
            neg_arc_limit = neg_arc_limit - (2*PI);
                  if (pos_arc_limit > 0)
                    pos_arc_limit = pos_arc_limit - (2*PI);
                  if (arc_foot > 0)
                    arc_foot = arc_foot - (2*PI);
                  if (arc_foot >= neg_arc_limit && arc_foot <= pos_arc_limit)
                    arcsegment_flag = 1;
  } // direction test ends here
 } // foot number test ends here
```

```
  } // intersection test ends here
 } // perpindicular test ends here

 if (arcsegment_flag != 1){
  arcsegment_flag = 0;
  } // nullifying parameters ends here

 return (arcsegment_flag);

} // arcint function ends here

// *************************NPS*and*PHRI************************************
//REVISED: .....
//TITLE:   segint()
//INPUT:   (x,y)body coordinates of a selected foot; direction; inner
//         endpoints; outer endpoints
//OUTPUT:  intercepts of line segment; mdistance to the intercepts from the
//         current foot position; flag to indicate whether an intercept
//                was found
//FUNCTION: This function determines whether there is an intercept of the
//                desired direction of the foot with a line segment defining a part
//         of the workspace.
// ************************AQUAROBOT************************************

int segint(double foot[], double direction, double inseg[], double outseg[],
        double intercepts[], double &mdistance)
{
 // initialization
 double a, b, c, d, theta_segment, theta_difference;
 double x_numerator, y_numerator, denominator;
 double x_intercept, y_intercept;
 double segment_test, CCW, CW, point_1, point_2;
 int linesegment_flag = 0;

 // find theta of line segment
 theta_segment = atan2(outseg[1] - inseg[1], outseg[0] - inseg[0]);
 theta_difference = direction - theta_segment;

 // if there is no difference between direction ray and the orientation of
 // the line segment being tested, then they are parallel; no intersection
 if ( theta_difference != 0.0
   && theta_difference != PI
   && theta_difference != -PI){
  b = (inseg[0]*sin(theta_segment)) - (inseg[1]*cos(theta_segment));
  a = (foot[0]*sin(direction)) - (foot[1]*cos(direction));
  x_numerator = (-cos(theta_segment)*a) + (cos(direction)*b);
  y_numerator = (sin(direction)*b) - (sin(theta_segment)*a);
  denominator = sin(theta_segment - direction);
  x_intercept = x_numerator / denominator;
  y_intercept = y_numerator / denominator;
  intercepts[0] = x_intercept;
  intercepts[1] = y_intercept;

  // determine if the direction ray intersects the line segment
  CCW = (inseg[0]*cos(theta_segment) + inseg[1]*sin(theta_segment));
  CW = (outseg[0]*cos(theta_segment) + outseg[1]*sin(theta_segment));
  segment_test = (x_intercept*cos(theta_segment)
                        + y_intercept*sin(theta_segment));
  if ((CCW <= segment_test ) && (CW >= segment_test)){  // if true, intersection

   // determine if the orientation is correct
   point_2 = (x_intercept*cos(direction) + y_intercept*sin(direction));
   point_1 = (foot[0]*cos(direction) + foot[1]*sin(direction));

   if (point_2 > point_1){
          // correct orientation
          linesegment_flag = 1;
```

```
            c = (y_intercept - foot[1])*(y_intercept - foot[1]);
            d = (x_intercept - foot[0])*(x_intercept - foot[0]);
            mdistance = sqrt(c + d);
    } // orientation test ends here

   } // line segment intersection test ends here

  } // line intersection test ends here

  // if one of the tests fails, there is no intersection
  if (linesegment_flag != 1){
   linesegment_flag = 0;
  } // nullifying parameters ends here

  return (linesegment_flag);

} // segint function ends here

#ifndef ARFUNC_H
#define ARFUNC_H

// ********************************************************************************
// FILENAME: arfunc.H
// PURPOSE:
// COMMENT: include file
// ********************************************************************************

#include "arconst.H"
#include "Kinematics.H"

extern
double min(double, double);

extern
double max(double, double);

extern
void ellipse(WalkParameter &, double *, double *);

extern
void kinematics(double*, double*, double*, double*, double*);

extern
void inv  inematics(double*, double*);

extern
void world_body(double*, double*, double*);

extern
void body_world(double*, double*, double*);

extern
int arcint(double*, double, double*, double, double*, double*,int,double*, double &);

extern
int segint(double*, double, double*, double*, double*, double &, int);

extern
double maxdistance_25cmfoot(Next_Motion &, WalkParameter &);

extern
double maxdistance_45cmfoot(Next_Motion &, WalkParameter &);

#endif

// ********************************************************************************
// FILENAME: artpgait.C
```

```
// PURPOSE:  tripod gait using ellipse trajectory
// COMMENT:  fprintf statement is added to make know the motor speed.
// ********************************************************************

#include <stdio.h>
#include <stream.h>
#include <math.h>
#include <stdlib.h>

#include "arconst.H"
#include "arfunc.H"
#include "artpgait.H"
#include "Kinematics.H"

// ********************************************************************
// FUNCTION: get_goal()
// ********************************************************************
void get_goal(double goal[])
{
 printf("Goal point( x  y )==>");
 scanf("%lf %lf", &goal[XW], &goal[YW]);

}// end of get_goal()

// ********************************************************************
// FUNCTION: get_menu()
// ********************************************************************
int get_menu(int menu)
{
 printf("0...Discrete Motion\n");
 printf("1...Continuous Motion ==>");
 scanf("%d", &menu);

 return (menu);

}// end of get_menu()

// ********************************************************************
// FUNCTION: get_footchoice()        (added 06May93)
// ********************************************************************
int get_footchoice(int footsize)
{
 printf("0...25 cm diameter footpad\n");
 printf("1...45 cm diameter footpad ==>");
 scanf("%d", &footsize);

 return (footsize);

}// end of get_footchoice()

// ********************************************************************
// FUNCTION: gait_algorithm()
// ********************************************************************
void gait_algorithm(Next_Motion &actual,
                    double goal[],
                    int menu,
                    int footsize) // selecting discrete or continuous motion
{
 static WalkParameter wp;
 static Flag flag;
 static int counter;


 switch (menu) {
 case 0:  // discrete motion
          /* set flag */
          disc_set_flag(actual, wp, goal, flag);
```

```
            /* gait_planning */
            if (flag.phaseEnd == 1) {
             disc_gait_planning(actual, wp, flag, goal, footsize);
            }

            /* Motion Planning */
            disc_tripod_motion(actual, wp, flag);

            break;

 case 1:  // continuous motion
            /* set flag */
            cont_set_flag(actual, wp, goal, flag);

            /* gait_planning */
            if (flag.phaseEnd == 1) {
             cont_gait_planning(actual, wp, flag, goal);
            }

            /* Motion Planning */
            cont_tripod_motion(actual, wp, flag);

            break;

 default:
            break;
 }

 // print_walkpara(wp);

}// End of gait_algorithm()

// ********************************************************************************
// FUNCTION: robot_model()
// ********************************************************************************
void robot_model(Next_Motion &nm)
{
 double euler[3];            // Euler angle
 double body[3];             // Body Position
 double jangle[LEG][JOINT];    // Joint angle
 double jpoint[LEG][JOINT][XYZ]; // Joint Position
 int leg, joint, axis;


 /* Initialize */
 euler[0] = nm.body_center_coord[0];
 euler[1] = nm.body_center_coord[1];
 euler[2] = nm.body_center_coord[2];
 body[XW] = nm.body_center_coord[3];
 body[YW] = nm.body_center_coord[4];
 body[ZW] = nm.body_center_coord[5];

 /* joint angle initialize */
 for (leg=LEG1; leg<=LEG6; leg++) {
  for (joint=CB; joint<=FOOT; joint++) {
   jangle[leg][joint] = nm.inbd_joint_angle[leg][joint];
                        //Tricky part! inbd_joint_angle[][HIP->KNEE2]{HIP,KN1,KN2,FT}
                        //Tricky part!        jangle[][CB ->KNEE2]{CB,HIP,KN1,KN2,FT}
  }
 }

 for (leg=LEG1; leg<=LEG6; leg++) {
            /* Kinematics computation for GNU plot data file */
            kinematics(jangle[leg], jpoint[leg][HIP], jpoint[leg][KNEE1],
                                          jpoint[leg][KNEE2], jpoint[leg][FOOT]);
```

```
            for(joint=HIP; joint<=FOOT; joint++){
              /* Coordinate Transformation(WORLD<-BODY) */
              world_body(euler, body, jpoint[leg][joint]);
            }
}

/* set Next_Motion */
for (axis=XW; axis<=ZW; axis++) {
            nm.foot_1_coord[axis] = jpoint[LEG1][FOOT][axis];
            nm.foot_2_coord[axis] = jpoint[LEG2][FOOT][axis];
            nm.foot_3_coord[axis] = jpoint[LEG3][FOOT][axis];
            nm.foot_4_coord[axis] = jpoint[LEG4][FOOT][axis];
            nm.foot_5_coord[axis] = jpoint[LEG5][FOOT][axis];
            nm.foot_6_coord[axis] = jpoint[LEG6][FOOT][axis];
}

/* set contact flag. */
if (nm.foot_1_coord[ZW] >= -0.01) {
            nm.leg_contact_flag[LEG1] = 1; // set contact flag(Leg1)
 nm.foot_1_coord[ZW] = 0.0;
}
else {
            nm.leg_contact_flag[LEG1] = 0; // reset contact flag
}

if (nm.foot_2_coord[ZW] >= -0.01) {
 nm.leg_contact_flag[LEG2] = 1; // set contact flag(Leg2)
 nm.foot_2_coord[ZW] = 0.0;
}
else {
 nm.leg_contact_flag[LEG2] = 0; // reset contact flag
}

if (nm.foot_3_coord[ZW] >= -0.01) {
            nm.leg_contact_flag[LEG3] = 1; // set contact flag(Leg3)
 nm.foot_3_coord[ZW] = 0.0;
}
else {
            nm.leg_contact_flag[LEG3] = 0; // reset contact flag
}

if (nm.foot_4_coord[ZW] >= -0.01) {
 nm.leg_contact_flag[LEG4] = 1; // set contact flag(Leg4)
}
else {
 nm.leg_contact_flag[LEG4] = 0; // reset contact flag
}

if (nm.foot_5_coord[ZW] >= -0.01) {
             nm.leg_contact_flag[LEG5] = 1; // set contact flag(Leg5)
 nm.foot_5_coord[ZW] = 0.0;
}
else {
             nm.leg_contact_flag[LEG5] = 0; // reset contact flag
}

if (nm.foot_6_coord[ZW] >= -0.01) {
 nm.leg_contact_flag[LEG6] = 1; // set contact flag(Leg6)
 nm.foot_6_coord[ZW] = 0.0;
}
else {
 nm.leg_contact_flag[LEG6] = 0; // reset contact flag
}

}// End of robot_model()

// ***************************************************************************
```

```
// TITLE:    disc_set_flag();
// FUNCTION: to set flags.
// INPUT:    phase, Next_Motion class, Flag class.
// OUTPUT:   Flag class
// CALLED BY: gait_algorithm()
// CALLS:    None
// COMMENT:  for discrete motion.
// ***************************************************************************
void disc_set_flag(Next_Motion &nm,
                                                    WalkParameter &wp,
                                                    double goal[],
                                                    Flag &flag)

{
 double distance;

 /* set phase end flag */
 if ( (wp.phase == 0) &&  // TP0 phase
                          (nm.leg_contact_flag[LEG1] == 1) && // TP0 contuct
                          (flag.phaseEnd == 0)) {
            flag.phaseEnd = 1;
 }
 else if ( (wp.phase == 2) &&  // TP1 phase
                          (nm.leg_contact_flag[LEG4] == 1) && // TP1 contuct
                          (flag.phaseEnd == 0)) {
            flag.phaseEnd = 1;
 }
 else if ( (wp.phase == BODY0) &&  // BODY phase
                          (fabs(nm.body_center_coord[3] - (nm.foot_1_coord[X] - TripodSize))
                          < 1.0) &&
                          (flag.phaseEnd == 0) ) {
  flag.phaseEnd = 1;
 }
 else if ( (wp.phase == BODY1) &&  // BODY phase
                          (fabs(nm.body_center_coord[3] - (nm.foot_4_coord[X] + TripodSize))
                          < 1.0) &&
                          (flag.phaseEnd == 0) ) {
  flag.phaseEnd = 1;
 }
 else {
            flag.phaseEnd = 0;
 }

 /* Set GOAL flag */
 distance = sqrt( (nm.body_center_coord[3] - goal[XW])
                          *(nm.body_center_coord[3] - goal[XW])
                          + (nm.body_center_coord[4] - goal[YW])
                          *(nm.body_center_coord[4] - goal[YW]) );

 if (distance <= wp.bodyspeed) {
  flag.phaseEnd = 1;
  flag.goal = 1;
 }

}// End of disc_set_flag()

// ***************************************************************************
// TITLE:    disc_gait_planning()
// FUNCTION: Gait planning. This function determins walking parameters.
// INPUT:    motion phase, WalkParameter class, Flag class.
// OUTPUT:   WalkParameter class.
// RETURN:   motion phase.
// CALLED BY: main().
// CALLS:    None
// COMMENT:  for discrete motion
// ***************************************************************************
void disc_gait_planning(Next_Motion &nm,  //
                                          WalkParameter &wp,//
```

```
                                        Flag &flag,     //
                                        double goal[XY],
                                        int footsize) // goal position
{
 static int counter = 0; // to count phase number
 double distance;       // distance between START and GOAL

 /* goal case */
 if (flag.goal == 1) {
          exit(0);
 }

 /* to find direction */
 wp.direction = atan2(goal[YW] - nm.body_center_coord[4],
        goal[XW] - nm.body_center_coord[3])/(DR); // in degrees

 /* to determine a stride */
 if (counter == 0) {

          printf("\n");
          printf("PHASE0 = %d\n", wp.phase);

          if (footsize == 0) {
                   wp.stride = maxdistance_25cmfoot(nm, wp);
          }
          else {
                   wp.stride = maxdistance_45cmfoot(nm, wp);
          }
 }
 else {

          /* increment phase */
          wp.phase = wp.phase + 1;
          wp.phase = wp.phase % 4;
          printf("PHASE = %d\n", wp.phase);

          if (footsize == 0) {
                   if ( (wp.phase == 0) || (wp.phase == 2) ) {
                     wp.stride = maxdistance_25cmfoot(nm, wp);
                   }
    else {
           if (counter == 1) {
                            wp.bodyspeed = 0.5;
                    }
                    else {
                            wp.bodyspeed = 0.5;
                    }
    }
          }
          else {
                   // this area is for 45cm footpad; duplicate 25cm footpad routines
          }
 }

 counter++;

 /* to find distance between current position and GOAL */
 distance = sqrt( (goal[XW] - nm.body_center_coord[3])
          *(goal[XW] - nm.body_center_coord[3])
         + (goal[YW] - nm.body_center_coord[4])
          *(goal[YW] - nm.body_center_coord[4]));

 /* handling in case of robot is near the goal */
 if (distance < (wp.stride/2.0)) {
  wp.stride = 2.0*distance;
 }
```

```
}/* End of disc_gait_planning() */

// ***************************************************************************
// FUNCTION: disc_tripod_motion()
// PURPOSE:  discrete tripod motion.
//         to compute one incremental motion using ellipse trajectory.
//         to calculate one incremented joint statuses
// INPUT:    class Next_Motion,
// OUTPUT:   class Next_Motion
// COMMENT:  called by motion_planning().
// ***************************************************************************
void disc_tripod_motion(Next_Motion &nm,
                                              WalkParameter &wp,
                                              Flag &flag)
{
 double euler[ThreeDim], oldeuler[ThreeDim];// Euler angles
 double deuler[ThreeDim];              // Change in Euler angle
 double body[ThreeDim],  oldbody[ThreeDim]; // CB position on World
 double dbody[ThreeDim];               // Change in BODY position
 double oldjangle[LEG][JOINT];         // old joint angles
 double jangle[LEG][JOINT];            // joint angles
 double jpoint[LEG][JOINT][ThreeDim];     // joint positions
 static double footprint[LEG][ThreeDim];   // foot print(contacted  point)
 double dbm;
 int leg, joint, axis;

 /* Initialization */
 for (axis=XW; axis<=ZW; axis++) {
  /* euler angles are not needed for ver.1.0 */
  euler[axis] = oldeuler[axis] = nm.body_center_coord[axis];
  body[axis]  = oldbody[axis]  = nm.body_center_coord[axis+3];

          /* Feet position initialize */
  jpoint[LEG1][FOOT][axis] = nm.foot_1_coord[axis];
  jpoint[LEG2][FOOT][axis] = nm.foot_2_coord[axis];
  jpoint[LEG3][FOOT][axis] = nm.foot_3_coord[axis];
  jpoint[LEG4][FOOT][axis] = nm.foot_4_coord[axis];
  jpoint[LEG5][FOOT][axis] = nm.foot_5_coord[axis];
  jpoint[LEG6][FOOT][axis] = nm.foot_6_coord[axis];

          /* footprint initialize */
  if (flag.phaseEnd == 1) {
                    if (nm.leg_contact_flag[LEG1] == 1)
                     footprint[LEG1][axis] = nm.foot_1_coord[axis];
                    if (nm.leg_contact_flag[LEG2] == 1)
                     footprint[LEG2][axis] = nm.foot_2_coord[axis];
                    if (nm.leg_contact_flag[LEG3] == 1)
                     footprint[LEG3][axis] = nm.foot_3_coord[axis];
                    if (nm.leg_contact_flag[LEG4] == 1)
                     footprint[LEG4][axis] = nm.foot_4_coord[axis];
                    if (nm.leg_contact_flag[LEG5] == 1)
                     footprint[LEG5][axis] = nm.foot_5_coord[axis];
                    if (nm.leg_contact_flag[LEG6] == 1)
                     footprint[LEG6][axis] = nm.foot_6_coord[axis];
  }
 }

 /* Joint angle initialize */
 for (leg=LEG1; leg<=LEG6; leg++) {
  for (joint=CB; joint<=FOOT; joint++) {
   jangle[leg][joint] = nm.inbd_joint_angle[leg][joint];
   oldjangle[leg][joint] = jangle[leg][joint];
                    //Tricky part! inbd_joint_angle[][HIP->KNEE2]{HIP,KN1,KN2}
                    //Tricky part!        jangle[][CB ->KNEE2]{CB,HIP,KN1,KN2}
  }
 }
```

```
if ((wp.phase%2) == 1) { // if BODY phase, then increment body position
        dbm = wp.bodyspeed;

        /* to increment body position */
        body[XW] += dbm*cos(wp.direction*DR);
        body[YW] += dbm*sin(wp.direction*DR);
        body[ZW] += 0.0;
}

/* to calculate joint angles */
for (leg=LEG1; leg<=LEG6; leg++) {

        if (  ((wp.phase == 0) && ((leg%2) == 0))     // tp0 AND odd leg#
                        ||((wp.phase == 2) && ((leg%2) == 1)) ){   // tp1 AND even leg#

                /* compute FOOT position on an ellipse curve respect to WORLD */
                ellipse(wp, footprint[leg], jpoint[leg][FOOT]);

                /* Foot Contact Condition.*/
                if (jpoint[leg][FOOT][ZW] > 0.0) {
                 jpoint[leg][FOOT][XW] = footprint[leg][XW]
                                + wp.stride*cos(wp.direction*DR);
                 jpoint[leg][FOOT][YW] = footprint[leg][YW]
                                + wp.stride*sin(wp.direction*DR);
                 jpoint[leg][FOOT][ZW] = footprint[leg][ZW];  //0.0;
                }
        }

        /* Coordinates Transformation(BODY<-WORLD) */
        body_world(euler, body, jpoint[leg][FOOT]);

 /* Inverse Kinematics */
 inv_kinematics(jpoint[leg][FOOT], jangle[leg]);
}

/* set joint angle */
for(joint=CB; joint<=KNEE2; joint++){
        for(leg=LEG1; leg<=LEG6; leg++){
         nm.inbd_joint_angle[leg][joint] = jangle[leg][joint];
         //Tricky part! inbd_joint_angle[][HIP->KNEE2]{HIP,KN1,KN2,FT}
         //Tricky part!        jangle[][CB ->KNEE2]{CB,HIP,KN1,KN2,FT}
 }
}

/* to set next status */
nm.body_center_coord[3] = body[XW];
nm.body_center_coord[4] = body[YW];
nm.body_center_coord[5] = body[ZW];

/* to calculate change in body position and euler angles */
for (axis=XW; axis<=ZW; axis++) {
        deuler[axis] = euler[axis] - oldeuler[axis];
        dbody[axis] = body[axis] - oldbody[axis];
}

/* to calculate change in BODY position */
nm.bodymotion[0] = deuler[AZIMUTH];
nm.bodymotion[1] = deuler[ELEVATION];
nm.bodymotion[2] = deuler[ROLL];
nm.bodymotion[3] = dbody[XW];
nm.bodymotion[4] = dbody[YW];
nm.bodymotion[5] = dbody[ZW];


/* to calculate change in joint angles */
for(joint=HIP; joint<=KNEE2; joint++){
 nm.leg1motion[joint-1] = jangle[LEG1][joint] - oldjangle[LEG1][joint];
```

```
    nm.leg2motion[joint-1] = jangle[LEG2][joint] - oldjangle[LEG2][joint];
    nm.leg3motion[joint-1] = jangle[LEG3][joint] - oldjangle[LEG3][joint];
    nm.leg4motion[joint-1] = jangle[LEG4][joint] - oldjangle[LEG4][joint];
    nm.leg5motion[joint-1] = jangle[LEG5][joint] - oldjangle[LEG5][joint];
    nm.leg6motion[joint-1] = jangle[LEG6][joint] - oldjangle[LEG6][joint];
  }

}/* End of disc_tripod_motion */

// ****************************************************************************
// TITLE:    cont_set_flag();
// FUNCTION: to set flags.
// INPUT:    phase, Next_Motion class, Flag class.
// OUTPUT:   Flag class
// CALLED BY: gait_algorithm()
// CALLS:    None
// COMMENT:  for continuous motion.
// ****************************************************************************
void cont_set_flag(Next_Motion &nm,
                   WalkParameter &wp,
                   double goal[],
                   Flag &flag)
{
 double distance;

 /* set phase end flag */
 if ( (wp.phase == 0) && // TP0 phase
      (nm.leg_contact_flag[LEG1] == 1) && // TP0 contuct
      (flag.phaseEnd == 0)) {
     flag.phaseEnd = 1;
 }
 else if ( (wp.phase == 1) && // TP1 phase
           (nm.leg_contact_flag[LEG4] == 1) && // TP1 contuct
           (flag.phaseEnd == 0)) {
     flag.phaseEnd = 1;
 }
 else {
     flag.phaseEnd = 0;
 }

 /* Set GOAL flag */
 distance = sqrt( (nm.body_center_coord[3] - goal[XW])
                  *(nm.body_center_coord[3] - goal[XW])
                 + (nm.body_center_coord[4] - goal[YW])
                  *(nm.body_center_coord[4] - goal[YW]) );

 if (distance <= wp.bodyspeed) {
   flag.phaseEnd = 1;
   flag.goal = 1;
 }

}// End of set_flag()

// ****************************************************************************
// TITLE:    gait_planning()
// FUNCTION: Gait planning module. This function determins walking parameters.
// INPUT:    motion phase, WalkParameter class, Flag class.
// OUTPUT:   WalkParameter class.
// RETURN:   motion phase.
// CALLED BY: main().
// CALLS:    None
// COMMENT:  for continuous motion
// ****************************************************************************
void cont_gait_planning(Next_Motion &nm, //
                        WalkParameter &wp,//
                        Flag &flag,     //
                        double goal[XY]) // goal position
```

```
{
 static int counter=0; // to count phase number
 double distance;      // distance between START and GOAL

 /* goal case */
 if (flag.goal == 1) {
          exit(0);
 }

 /* to determine a stride */
 if (counter == 0) {
  wp.stride = DefaultSTRIDE/2.0;
           wp.bodyspeed=((DefaultSTRIDE/2.0)/FINE)/2.0;
 }
 else {
  wp.stride = DefaultSTRIDE;
  wp.bodyspeed=(DefaultSTRIDE/2.0)/FINE;

          /* increment phase */
           wp.phase = wp.phase + 1;
           wp.phase = wp.phase % 2; // continuous:2, discrete:4
 }
 counter++;

 /* to find distance between current position and GOAL */
 distance = sqrt( (goal[XW] - nm.body_center_coord[3])
          *(goal[XW] - nm.body_center_coord[3])
          + (goal[YW] - nm.body_center_coord[4])
          *(goal[YW] - nm.body_center_coord[4]));

 /* handling in case of robot is near the goal */
 if (distance < (wp.stride/2.0)) {
  wp.stride = 2.0*distance;
 }

 /* to find direction */
 wp.direction = atan2(goal[YW] - nm.body_center_coord[4],
                      goal[XW] - nm.body_center_coord[3])/(DR); // degree

}/* End of gait_planning() */

// ***************************************************************************
// FUNCTION: cont_tripod_motion()
// PURPOSE:  continuous tripod motion.
//       to compute one incremental motion using ellipse trajectory.
//       to calculate one incremented joint statuses
// INPUT:   class Next_Motion,
// OUTPUT:  class Next_Motion
// COMMENT: called by gait_planning().
// ***************************************************************************
void cont_tripod_motion(Next_Motion &nm,
                        WalkParameter &wp,
                        Flag &flag)
{
 double euler[ThreeDim], oldeuler[ThreeDim];// Euler angles
 double deuler[ThreeDim];              // Change in Euler angle
 double body[ThreeDim], oldbody[ThreeDim]; // CB position on World
 double dbody[ThreeDim];               // Change in BODY position
 double oldjangle[LEG][JOINT];         // old joint angles
 double jangle[LEG][JOINT];            // joint angles
 double jpoint[LEG][JOINT][ThreeDim];  // joint positions
 static double footprint[LEG][ThreeDim];  // foot print(contacted point)
 double dbm;
 int leg, joint, axis;

 /* Initialize */
 for (axis=XW; axis<=ZW; axis++) {
```

```
/* euler angles are not needed for ver.1.0 */
euler[axis] = oldeuler[axis] = nm.body_center_coord[axis];
body[axis]  = oldbody[axis]  = nm.body_center_coord[axis+3];

        /* Feet position initialize */
jpoint[LEG1][FOOT][axis] = nm.foot_1_coord[axis];
jpoint[LEG2][FOOT][axis] = nm.foot_2_coord[axis];
jpoint[LEG3][FOOT][axis] = nm.foot_3_coord[axis];
jpoint[LEG4][FOOT][axis] = nm.foot_4_coord[axis];
jpoint[LEG5][FOOT][axis] = nm.foot_5_coord[axis];
jpoint[LEG6][FOOT][axis] = nm.foot_6_coord[axis];

        /* footprint initialize */
 if (flag.phaseEnd == 1) {
          if (nm.leg_contact_flag[LEG1] == 1)
                  footprint[LEG1][axis] = nm.foot_1_coord[axis];
          if (nm.leg_contact_flag[LEG2] == 1)
                  footprint[LEG2][axis] = nm.foot_2_coord[axis];
          if (nm.leg_contact_flag[LEG3] == 1)
                  footprint[LEG3][axis] = nm.foot_3_coord[axis];
          if (nm.leg_contact_flag[LEG4] == 1)
                  footprint[LEG4][axis] = nm.foot_4_coord[axis];
          if (nm.leg_contact_flag[LEG5] == 1)
                  footprint[LEG5][axis] = nm.foot_5_coord[axis];
          if (nm.leg_contact_flag[LEG6] == 1)
                  footprint[LEG6][axis] = nm.foot_6_coord[axis];
 }
}

/* Joint angle initialize */
for (leg=LEG1; leg<=LEG6; leg++) {
 for (joint=CB; joint<=FOOT; joint++) {
  jangle[leg][joint] = nm.inbd_joint_angle[leg][joint];
  oldjangle[leg][joint] = jangle[leg][joint];
                  //Tricky part! inbd_joint_angle[][HIP->KNEE2]{HIP,KN1,KN2,FT}
                  //Tricky part!        jangle[][CB ->KNEE2]{CB,HIP,KN1,KN2,FT}
 }
}

dbm = wp.bodyspeed;

/* to increment body position */
body[XW] += dbm*cos(wp.direction*DR);
body[YW] += dbm*sin(wp.direction*DR);
body[ZW] += 0.0;

/* to set next status */
nm.body_center_coord[3] = body[XW];
nm.body_center_coord[4] = body[YW];
nm.body_center_coord[5] = body[ZW];

/* to calculate change in body position and euler angles */
for (axis=XW; axis<=ZW; axis++) {
          deuler[axis] = euler[axis] - oldeuler[axis];
          dbody[axis] = body[axis] - oldbody[axis];
}

/* to calculate change in BODY position */
nm.bodymotion[0] = deuler[AZIMUTH];
nm.bodymotion[1] = deuler[ELEVATION];
nm.bodymotion[2] = deuler[ROLL];
nm.bodymotion[3] = dbody[XW];
nm.bodymotion[4] = dbody[YW];
nm.bodymotion[5] = dbody[ZW];

/* to calculate joint angles */
for (leg=LEG1; leg<=LEG6; leg++) {
```

```
        if ( ((wp.phase == 0) && ((leg%2) == 0))     // tp0 AND odd leg#
    ||((wp.phase == 1) && ((leg%2) == 1)) ){   // tp1 AND even leg#

            /* compute FOOT position on an ellipse curve respect to WORLD */
            ellipse(wp, footprint[leg], jpoint[leg][FOOT]);

            /* Foot Contact Condition.*/ // Terrain Model == flat
            if (jpoint[leg][FOOT][ZW] > 0.0) {
                    jpoint[leg][FOOT][XW] = footprint[leg][XW]
                + wp.stride*cos(wp.direction*DR);
                    jpoint[leg][FOOT][YW] = footprint[leg][YW]
                + wp.stride*sin(wp.direction*DR);
                    jpoint[leg][FOOT][ZW] = footprint[leg][ZW];  //0.0;
            }
           }

           /* Coordinates Transformation(BODY -> WORLD) */
           body_world(euler, body, jpoint[leg][FOOT]);

  /* Inverse Kinematics */
  inv_kinematics(jpoint[leg][FOOT], jangle[leg]);
 }

 /* set joint angle */
 for (joint=CB; joint<=KNEE2; joint++) {
           for (leg=LEG1; leg<=LEG6; leg++) {
   nm.inbd_joint_angle[leg][joint] = jangle[leg][joint];
                    //Tricky part! inbd_joint_angle[][HIP->KNEE2]{HIP,KN1,KN2,FT}
                    //Tricky part!        jangle[][CB ->KNEE2]{CB,HIP,KN1,KN2,FT}
  }
 }

 /* to calculate change in joint angles */
 for (joint=HIP; joint<=KNEE2; joint++) {
  nm.leg1motion[joint-1] = jangle[LEG1][joint] - oldjangle[LEG1][joint];
  nm.leg2motion[joint-1] = jangle[LEG2][joint] - oldjangle[LEG2][joint];
  nm.leg3motion[joint-1] = jangle[LEG3][joint] - oldjangle[LEG3][joint];
  nm.leg4motion[joint-1] = jangle[LEG4][joint] - oldjangle[LEG4][joint];
  nm.leg5motion[joint-1] = jangle[LEG5][joint] - oldjangle[LEG5][joint];
  nm.leg6motion[joint-1] = jangle[LEG6][joint] - oldjangle[LEG6][joint];
 }

}/* End of cont_tripod_motion */

// ***************************************************************************
// FUNCTION: ctp_foot() (Center of the Tripod -> FOOT)
// ***************************************************************************
void ctp_foot(int tp_no, double ctp[], double foot[][XYZ], double tripodsize)
{
 double ratio = 3.0/2.0;
 int leg;

 for (leg=LEG1; leg<=LEG6; leg++) {

           if ( ((tp_no == 0) && ((leg%2) == 0)) ||
                    ((tp_no == 1) && ((leg%2) == 1)) ) {
            foot[leg][XW] = ctp[XW] + tripodsize*cos(leg*60.0*DR);
            foot[leg][YW] = ctp[YW] + tripodsize*sin(leg*60.0*DR);
            foot[leg][ZW] = ctp[ZW];
           }
 }
}// End of ctp_foot()

// ***************************************************************************
// FUNCTION: foot_ctp() (FOOT -> Center of the Tripod)
// ***************************************************************************
```

```
void foot_ctp(int tp_no, double ctp[], double foot[][XYZ])
{

 ctp[X] = (foot[tp_no][X] + foot[tp_no+2][X] + foot[tp_no+4][X])/3.0;
 ctp[Y] = (foot[tp_no][Y] + foot[tp_no+2][Y] + foot[tp_no+4][Y])/3.0;
 ctp[Z] = (foot[tp_no][Z] + foot[tp_no+2][Z] + foot[tp_no+4][Z])/3.0;

}// End of foot_ctp()

// ***************************************************************************
// TITLE:    print_status()
// FUNCTION: tc print out Next_motion class variable status
// INPUTS:   class Next_Motion,
// OUTPUTS:  None.
// CALLED BY: MotionPlanning().
// CALLS:    None
// ***************************************************************************
void print_status(Next_Motion &nm)
{
 int leg;

 printf("\n---AQUAROBOT CURRENT STATUS---\n");
 printf("POSITION\n");
 printf("  BODY  Xw=%f Yw=%f Zw=%f\n", nm.body_center_coord[3],
                          nm.body_center_coord[4],
                          nm.body_center_coord[5]);
 printf("  FOOT1 Xw=%f Yw=%f Zw=%f\n", nm.foot_1_coord[XW],
                          nm.foot_1_coord[YW],
                          nm.foot_1_coord[ZW]);
 printf("  FOOT2 Xw=%f Yw=%f Zw=%f\n", nm.foot_2_coord[XW],
                          nm.foot_2_coord[YW],
                          nm.foot_2_coord[ZW]);
 printf("  FOOT3 Xw=%f Yw=%f Zw=%f\n", nm.foot_3_coord[XW],
                          nm.foot_3_coord[YW],
                          nm.foot_3_coord[ZW]);
 printf("  FOOT4 Xw=%f Yw=%f Zw=%f\n", nm.foot_4_coord[XW],
                          nm.foot_4_coord[YW],
                          nm.foot_4_coord[ZW]);
 printf("  FOOT5 Xw=%f Yw=%f Zw=%f\n", nm.foot_5_coord[XW],
                          nm.foot_5_coord[YW],
                          nm.foot_5_coord[ZW]);
 printf("  FOOT6 Xw=%f Yw=%f Zw=%f\n", nm.foot_6_coord[XW],
                          nm.foot_6_coord[YW],
                          nm.foot_6_coord[ZW]);

 printf("JOINT ANGLE \n" );
 for(leg=LEG1; leg<=LEG6; leg++){
  printf("  LEG%d HIP=%f K1=%f K2=%f\n", leg+1,
                          nm.inbd_joint_angle[leg][HIP],   //HIP
                          nm.inbd_joint_angle[leg][KNEE1], //KNEE1
                          nm.inbd_joint_angle[leg][KNEE2]);//KNEE2
 }

 printf("FOOT CONTACT FLAG \n");
 for(leg=LEG1; leg<=LEG6; leg++){
          printf("  leg%d=%d", leg+1, nm.leg_contact_flag[leg]);
 }
 printf("\n");

}// End of print_status()

// ***************************************************************************
// TITLE:    print_walkpara()
// FUNCTION: to print out WalkParameter class variables status
// INPUTS:   class Walkparameter.
// OUTPUTS:  None.
// CALLED BY: MotionPlanning().
```

```
// CALLS:    None
// ***************************************************************************
void print_walkpara(WalkParameter &wp)
{
 printf("\n---WALK PARAMETERS---\n");
 printf("PHASE=%d\n",wp.phase);
 printf("STRIDE=%f DIRECTION=%f\n",wp.stride, wp.direction);
 printf("TPSIZE=%f FOTHEIGHT=%f\n",wp.tripodsize, wp.footheight);
 printf("BODY SPEED=%f\n",wp.bodyspeed);

}// End of print_walkpara()

// END OF FILE "artpgait.C"

// ***************************************************************************
// FUNCTION: print_gnu_data()
// ***************************************************************************
void print_gnu_data(Next_Motion &nm)
{
 double euler[3];              // Euler angle
 double body[3];               // Body Position
 double jangle[LEG][JOINT];    // Joint angle
 double jpoint[LEG][JOINT][XYZ]; // Joint Position
 int leg, joint;

 /* print for GNU plot data file */
 FILE *fp0, *fp1, *fp2;
 fp0 = fopen("artp1.dat","a");
 fp1 = fopen("arxy.dat","w");
 fp2 = fopen("arstable.dat","w");

 /* Initialization */
 euler[0] = nm.body_center_coord[0];
 euler[1] = nm.body_center_coord[1];
 euler[2] = nm.body_center_coord[2];
 body[XW] = nm.body_center_coord[3];
 body[YW] = nm.body_center_coord[4];
 body[ZW] = nm.body_center_coord[5];

 /* joint angle initialize */
 for (leg=LEG1; leg<=LEG6; leg++) {
  for (joint=CB; joint<=FOOT; joint++) {
   jangle[leg][joint] = nm.inbd_joint_angle[leg][joint];
  }
 }

 for (leg=LEG1; leg<=LEG6; leg++) {

        /* Kinematics computation for GNU plot data file */
        kinematics(jangle[leg], jpoint[leg][HIP], jpoint[leg][KNEE1],
                                jpoint[leg][KNEE2], jpoint[leg][FOOT]);

        for(joint=HIP; joint<=FOOT; joint++){

         /* Coordinate Transformation(WORLD<-BODY) */
         world_body(euler, body, jpoint[leg][joint]);

         fprintf(fp0,"%f %f \n",jpoint[leg][joint][X],jpoint[leg][joint][Z]);
        }
        fprintf(fp0,"\n");
}
fprintf(fp1,"%f %f \n",jpoint[LEG1][FOOT][XW],jpoint[LEG1][FOOT][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG1][HIP][XW],jpoint[LEG1][HIP][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG2][HIP][XW],jpoint[LEG2][HIP][YW]);
fprintf(fp1,"\n");
fprintf(fp1,"%f %f \n",jpoint[LEG2][FOOT][XW],jpoint[LEG2][FOOT][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG2][HIP][XW],jpoint[LEG2][HIP][YW]);
```

```
fprintf(fp1,"%f %f \n",jpoint[LEG3][HIP][XW],jpoint[LEG3][HIP][YW]);
fprintf(fp1,"\n");
fprintf(fp1,"%f %f \n",jpoint[LEG3][FOOT][XW],jpoint[LEG3][FOOT][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG3][HIP][XW],jpoint[LEG3][HIP][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG4][HIP][XW],jpoint[LEG4][HIP][YW]);
fprintf(fp1,"\n");
fprintf(fp1,"%f %f \n",jpoint[LEG4][FOOT][XW],jpoint[LEG4][FOOT][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG4][HIP][XW],jpoint[LEG4][HIP][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG5][HIP][XW],jpoint[LEG5][HIP][YW]);
fprintf(fp1,"\n");
fprintf(fp1,"%f %f \n",jpoint[LEG5][FOOT][XW],jpoint[LEG5][FOOT][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG5][HIP][XW],jpoint[LEG5][HIP][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG6][HIP][XW],jpoint[LEG6][HIP][YW]);
fprintf(fp1,"\n");
fprintf(fp1,"%f %f \n",jpoint[LEG6][FOOT][XW],jpoint[LEG6][FOOT][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG6][HIP][XW],jpoint[LEG6][HIP][YW]);
fprintf(fp1,"%f %f \n",jpoint[LEG1][HIP][XW],jpoint[LEG1][HIP][YW]);
fprintf(fp1,"\n");
fprintf(fp1,"%f %f \n", body[XW], body[YW]);
fprintf(fp1,"\n");

fprintf(fp2,"%f  %f \n",jpoint[LEG1][FOOT][XW],jpoint[LEG1][FOOT][YW]);
fprintf(fp2,"%f  %f \n",jpoint[LEG3][FOOT][XW],jpoint[LEG3][FOOT][YW]);
fprintf(fp2,"%f  %f \n",jpoint[LEG5][FOOT][XW],jpoint[LEG5][FOOT][YW]);
fprintf(fp2,"%f  %f \n",jpoint[LEG1][FOOT][XW],jpoint[LEG1][FOOT][YW]);
fprintf(fp2,"\n");
fprintf(fp2,"%f  %f \n",jpoint[LEG2][FOOT][XW],jpoint[LEG2][FOOT][YW]);
fprintf(fp2,"%f  %f \n",jpoint[LEG4][FOOT][XW],jpoint[LEG4][FOOT][YW]);
fprintf(fp2,"%f  %f \n",jpoint[LEG6][FOOT][XW],jpoint[LEG6][FOOT][YW]);
fprintf(fp2,"%f  %f \n",jpoint[LEG2][FOOT][XW],jpoint[LEG2][FOOT][YW]);
fprintf(fp2,"\n");
fprintf(fp2,"%f  %f \n", body[XW], body[YW]);
fprintf(fp2,"\n");

fclose(fp0);
fclose(fp1);
fclose(fp2);

}/* End of print_gnu_data() */

// ****************************************************************************
// FUNCTION: print_joint_data()
// ****************************************************************************
void print_joint_data(Next_Motion &nm)
{
 /* print for GNU plot data file */
 FILE *fp11, *fp21, *fp31, *fp41, *fp51, *fp61;
 FILE *fp12, *fp22, *fp32, *fp42, *fp52, *fp62;
 FILE *fp13, *fp23, *fp33, *fp43, *fp53, *fp63;
 fp11 = fopen("leg1hip.dat","a");
 fp12 = fopen("leg1kn1.dat","a");
 fp13 = fopen("leg1kn2.dat","a");
 fp21 = fopen("leg2hip.dat","a");
 fp22 = fopen("leg2kn1.dat","a");
 fp23 = fopen("leg2kn2.dat","a");
 fp31 = fopen("leg3hip.dat","a");
 fp32 = fopen("leg3kn1.dat","a");
 fp33 = fopen("leg3kn2.dat","a");
 fp41 = fopen("leg4hip.dat","a");
 fp42 = fopen("leg4kn1.dat","a");
 fp43 = fopen("leg4kn2.dat","a");
 fp51 = fopen("leg5hip.dat","a");
 fp52 = fopen("leg5kn1.dat","a");
 fp53 = fopen("leg5kn2.dat","a");
 fp61 = fopen("leg6hip.dat","a");
 fp62 = fopen("leg6kn1.dat","a");
 fp63 = fopen("leg6kn2.dat","a");
```

```
  // print joint angle
  fprintf(fp11,"%f \n", nm.inbd_joint_angle[LEG1][HIP]);
  fprintf(fp21,"%f \n", nm.inbd_joint_angle[LEG2][HIP]);
  fprintf(fp31,"%f \n", nm.inbd_joint_angle[LEG3][HIP]);
  fprintf(fp41,"%f \n", nm.inbd_joint_angle[LEG4][HIP]);
  fprintf(fp51,"%f \n", nm.inbd_joint_angle[LEG5][HIP]);
  fprintf(fp61,"%f \n", nm.inbd_joint_angle[LEG6][HIP]);
  fprintf(fp12,"%f \n", nm.inbd_joint_angle[LEG1][KNEE1]);
  fprintf(fp22,"%f \n", nm.inbd_joint_angle[LEG2][KNEE1]);
  fprintf(fp32,"%f \n", nm.inbd_joint_angle[LEG3][KNEE1]);
  fprintf(fp42,"%f \n", nm.inbd_joint_angle[LEG4][KNEE1]);
  fprintf(fp52,"%f \n", nm.inbd_joint_angle[LEG5][KNEE1]);
  fprintf(fp62,"%f \n", nm.inbd_joint_angle[LEG6][KNEE1]);
  fprintf(fp13,"%f \n", nm.inbd_joint_angle[LEG1][KNEE2]);
  fprintf(fp23,"%f \n", nm.inbd_joint_angle[LEG2][KNEE2]);
  fprintf(fp33,"%f \n", nm.inbd_joint_angle[LEG3][KNEE2]);
  fprintf(fp43,"%f \n", nm.inbd_joint_angle[LEG4][KNEE2]);
  fprintf(fp53,"%f \n", nm.inbd_joint_angle[LEG5][KNEE2]);
  fprintf(fp63,"%f \n", nm.inbd_joint_angle[LEG6][KNEE2]);

  fclose(fp11); fclose(fp12); fclose(fp13);
  fclose(fp21); fclose(fp22); fclose(fp23);
  fclose(fp31); fclose(fp32); fclose(fp33);
  fclose(fp41); fclose(fp42); fclose(fp43);
  fclose(fp51); fclose(fp52); fclose(fp53);
  fclose(fp61); fclose(fp62); fclose(fp63);

}/* End of print_joint_data() */

#ifndef ARTPGAIT_H
#define ARTPGAIT_H

/*************************************************************************
FILENAME: artpgait.H
*************************************************************************/

#include "Kinematics.H"

extern
void get_goal(double []);

extern
int get_menu(int);

extern
int get_footchoice(int);

extern
void gait_algorithm(Next_Motion &, double [], int, int);

extern
void robot_model(Next_Motion &);

extern
void disc_gait_planning(Next_Motion &, WalkParameter &, Flag &, double [], int);

extern
void disc_set_flag(Next_Motion &, WalkParameter &, double [], Flag &);

extern
void disc_tripod_motion(Next_Motion &, WalkParameter &, Flag &);

extern
void cont_gait_planning(Next_Motion &, WalkParameter &, Flag &, double []);

extern
```

```
void cont_set_flag(Next_Motion &, WalkParameter &, double [], Flag &);

extern
void cont_tripod_motion(Next_Motion &, WalkParameter &, Flag &);

extern
void ctp_foot(int, double [], double [][XYZ], double);

extern
void foot_ctp(int, double [], double [][XYZ]);

extern
void print_status(Next_Motion &);

extern
void print_walkpara(WalkParameter &);

extern
void print_gnu_data(Next_Motion &);

extern
void print_joint_data(Next_Motion &);

#endif

// ****************************************************************************
// FILENAME: bot.C
// PURPOSE:  This file makes a stick aquarobot graphics
//           interactive design
//           It utilizes Kinematic functions to determine xyz
//           coordinates
// CONTAINS: functions shown in bot.h
// NOTE:     This is and IRIS 3D program written in C++
// UPDATE:   Last Update 1993 Apr 30 by Kenji Suzuki
// ****************************************************************************

#include "gl.h"  // graphics library
#include "device.h" // graphics library file

#include <stdio.h>  // C++ library

#include "bot.H"   // declaration file
#include "Link.H"
#include "AbRigid.H"
#include "MatrixMy.H"
#include "AbBody.H"
#include "Kinematics.H"
#include "AbLeg.H"


/* added by Kenji Suzuki 1993 Apr 21 */
#include "arconst.H"
#include "arfunc.H"
#include "artpgait.H"

main()
{

 // value returned from the event queue
 short value;
 long  mainmenu;
 long  hititem;
 FILE *ifp;
 ifp = fopen("bot.dat","r");

 double goal[XY];  // goal position, WORLD coordinates
 Next_Motion actual, command, temp;
```

```
int menu = 0;
int footsize = 0;

menu = get_menu(menu);
get_goal(goal);
get_footchoice(footsize);

// initialize the IRIS system
initialize();

// Create Pop Up Menus
mainmenu = makethemenus();

// make the robot from its pieces
AquarobotBody aquabody;
AquaLeg leg1(aquabody,0.0);
AquaLeg leg2(aquabody,60.0);
AquaLeg leg3(aquabody,120.0);
AquaLeg leg4(aquabody,180.0);
AquaLeg leg5(aquabody,240.0);
AquaLeg leg6(aquabody,300.0);

Return_Coordinates coord;
Passing_Items pass;
Next_Motion trans;
int ans;
ans = 0;
pass.legnum = 9;

coord = FindPositions(aquabody, leg1, leg2, leg3, leg4, leg5, leg6);
trans = TransferToGait(coord, aquabody);

while (TRUE) {

 // do we have something on the event queue?
 if (qtest()) {

                switch (qread(&value)) {

                case REDRAW:
                 reshapeviewport();
                 break;

                case MENUBUTTON:     // Menu selections
                 if (value == 1) {
                        hititem = dopup(mainmenu);
                        processmenuhit(hititem);
                 }
                 break;

                case AKEY:
                 // the gait algorithm is incorporated in this case
                 // actual = trans;
                 // Something wrong!
                 // "trans" does not give correct values.
                 // by Kenji  1993/4/30,15:00

                 temp = actual;

                 /* gait algorithm */
                 gait_algorithm(temp, goal, menu, footsize);

                 command = temp;

                 /* robot model */
                 robot_model(temp);
                 actual = temp;
```

```
                    trans = command;

    /*>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>*/

                    delaz = trans.bodymotion[0];
                    delel = trans.bodymotion[1];
                    delrol = trans.bodymotion[2];
                    delx = trans.bodymotion[3];
                    dely = trans.bodymotion[4];
                    delz = trans.bodymotion[5];

                    aquabody.MoveIncremental(delaz,delel,delrol,delx,dely,delz);
                    leg1.MoveIncremental(aquabody,trans.leg1motion[0],

trans.leg1motion[1],trans.leg1motion[2]);
                    leg2.MoveIncremental(aquabody,trans.leg2motion[0],

trans.leg2motion[1],trans.leg2motion[2]);
                    leg3.MoveIncremental(aquabody,trans.leg3motion[0],

trans.leg3motion[1],trans.leg3motion[2]);
                    leg4.MoveIncremental(aquabody,trans.leg4motion[0],

trans.leg4motion[1],trans.leg4motion[2]);
                    leg5.MoveIncremental(aquabody,trans.leg5motion[0],

trans.leg5motion[1],trans.leg5motion[2]);
                    leg6.MoveIncremental(aquabody,trans.leg6motion[0],

trans.leg6motion[1],trans.leg6motion[2]);

                    coord = FindPositions(aquabody,leg1,leg2,leg3,leg4,leg5,leg6);
                    trans = TransferToGait(coord,aquabody);
                    break;

                   case RKEY:
                    rewind(ifp);
                    break;

                   } // end switch on event queue item

               } // endif qtest()


               color(BLUE);  // background color
               clear();

               // turn on Z-buffering
               zbuffer(TRUE);

               // clear the z-buffer
               zclear();

               buildnonmovingviewingmatrix(vx,vy,vz,rfx,rfy,rfz);
               drawaqua((coord.bodyc),(coord.leg1c),(coord.leg2c),(coord.leg3c),
                                   (coord.leg4c),(coord.leg5c),(coord.leg6c));

               /* turn z-buffering off */
               zbuffer(FALSE);

               /* change the buffers ... */
               swapbuffers();

               qenter(AKEY,1);

    }
```

```
} // end of main

// ********************************************************
// FUNCTION: INITIALIZE()
// ********************************************************
void initialize()
{

 // set up the preferred aspect ratio
 keepaspect(XMAXSCREEN+1,YMAXSCREEN+1);

 // set up window size
// prefposition(700.0,1200.0,200.0,700.0);
// prefposition(200.0, 1300.0, 0.0, 900.0);
 prefposition(200.0, 750.0, 500.0, 900.0); // for video converter screen

 // open a window for the program
 winopen("AquaRobot");

 // make a title
 wintitle("AquaRobot");

 // put the IRIS into double buffer mode
 doublebuffer();

 // configure the IRIS (means use the above command settings)
 gconfig();

 /* queue the  devices */
 qdevice(REDRAW);
 qdevice(AKEY);
 qdevice(PKEY);
 qdevice(RKEY);
 qdevice(MENUBUTTON);

 vx =  0.0;
 vy =  400.0;
 vz =  800.0;

 rfx = 0.0;
 rfy = 0.0;
 rfz = 0.0;

}


/*****************************************************************************/
/* Function Make_the_Menus                                                   */
/*****************************************************************************/
long makethemenus()
{
 long topmenu;

 long cameramenu;
 /* camera views */
 cameramenu = newpup();
 addtopup(cameramenu,"Camera View %t ");
 addtopup(cameramenu,"ABOVE %x2 | LEG5-6 VIEW %x3");
 addtopup(cameramenu,"LEG1 VIEW  %x4 | LEG4 VIEW %x5");
 addtopup(cameramenu, "LEG2-3 VIEW %x9");

 // build the top level menu
 topmenu= defpup("Roll Off Side %t| Camera %x1 %m|FileRead %x6 |ResetFile %x8|KeybdRead %x7|Reset %x14| Exit
%x15",cameramenu);

 // return the name of this menu
```

```
 return(topmenu);

}

/*************************************************************************/
/* Function Process_Menu_Hit                                             */
/*************************************************************************/
void processmenuhit(long hititem)
{
 switch (hititem){
 case CAMERA: /* top slide off menu */
            break;

 case ABOVE:  // Graphics Coordinates
            vx =   0.0 + rfx;
            vy = 1000.0 + rfy;
            vz =   0.0 + rfz;
            break;

 case LEG56_VIEW:  /* View from the behind robot */
            vx =   0.0 + rfx;
            vy =  400.0 + rfy;
            vz = -800.0 + rfz;
            break;

 case LEG1_VIEW:
            vx =  800.0 + rfx;
            vy =  400.0 + rfy;
            vz =   0.0 + rfz;
            break;

 case LEG4_VIEW:
            vx = -800.0 + rfx;
            vy =  400.0 + rfy;
            vz =   0.0 + rfz;
            break;

 case FILEREAD:
            qenter(PKEY,1);
            break;

 case RESETFILE:
            qenter(RKEY,1);
            break;

 case LEG23_VIEW:
            vx =   0.0 + rfx;
            vy =  400.0 + rfy;
            vz =  800.0 + rfz;

 case RESET:
            vx =  0.0;
            vy =  400.0;
            vz =  800.0;
            break;

 case EXIT:
            exit(0);
            break;
 } // End Switch


}
/* for objects that are in the same coordinate system but aren't moving
with the continuous rotations/translations/scalings, we use this
routine ...
*/
```

```
// ***********************************************************
// FUNCTION: BUILDNONMOVINGVIEWINGMATRIX
// ***********************************************************
void buildnonmovingviewingmatrix(float vx,float vy,float vz,
                                                                    float
refx,float refy,float refz)
{

 // we must call loadunit before we get the projection
 //  and viewing stuff...

 loadunit();

 // just call the perspective + viewing matrices
 projectionandviewingmatrix(vx,vy,vz,refx,refy,refz);

}

/* put up the projection and viewing matrix */
// ***********************************************************
// FUNCTION: PROJECTIONANDVIEWINGMATRIX
// ***********************************************************
void projectionandviewingmatrix(float vx,float vy,float vz,
                        float refx,float refy,float refz)
{

 // perspective projection 3D for the world coord sys
 // the near and far values are distances from the viewer
 //  to the near and far clipping planes.
 //   We are at (vx,vy,vz) and looking towards
 //            the center point of the object..
 //             (towards (refx,refy,refz)).


 perspective(450,1.25,NEARCLIPPING,FARCLIPPING);
 lookat(vx,vy,vz,refx,refy,refz,0);

}


// this routine loads a unit matrix onto the top of the stack
void loadunit()
{
 static float  un[4][4] = {   1.0, 0.0, 0.0, 0.0,
                                                                    0.0, 1.0, 0.0, 0.0,
                                                                    0.0, 0.0, 1.0, 0.0,
                                                                    0.0, 0.0, 0.0, 1.0 };


 // load the matrix
 loadmatrix(un);

}

// ***********************************************************
//  AQUA DRAWING
// ***********************************************************
void drawaqua(double *bodyc,
                                    double *leg1c, double *leg2c, double *leg3c,
                                    double *leg4c, double *leg5c, double *leg6c)
{
 color(WHITE);
 linewidth(3);

 // NOTE!!!!!!!!!
 // +x to right, +y up, +z out of screen ->for graphics
```

```
// +x out of leg1, +y out of screen, -z down->for aquarobot

//      x    z      y
move(bodyc[3],-bodyc[5],bodyc[4]);
draw(bodyc[6],-bodyc[8],bodyc[7]);
draw(bodyc[9],-bodyc[11],bodyc[10]);
draw(bodyc[12],-bodyc[14],bodyc[13]);
draw(bodyc[15],-bodyc[17],bodyc[16]);
draw(bodyc[18],-bodyc[20],bodyc[19]);
draw(bodyc[3],-bodyc[5],bodyc[4]);

// draws a line from body center to leg1 joint !
linewidth(1);
move(bodyc[0],-bodyc[2],bodyc[1]);
draw(bodyc[3],-bodyc[5],bodyc[4]);

// draws leg1
color(YELLOW);
linewidth(5);
//      x      z      y
move(leg1c[0],-leg1c[2],leg1c[1]);
draw(leg1c[3],-leg1c[5],leg1c[4]);
draw(leg1c[6],-leg1c[8],leg1c[7]);
draw(leg1c[9],-leg1c[11],leg1c[10]);

// draws leg2
color(GREEN);
linewidth(5);
move(leg2c[0],-leg2c[2],leg2c[1]);
draw(leg2c[3],-leg2c[5],leg2c[4]);
draw(leg2c[6],-leg2c[8],leg2c[7]);
draw(leg2c[9],-leg2c[11],leg2c[10]);

// draws leg3
color(GREEN);
linewidth(5);
move(leg3c[0],-leg3c[2],leg3c[1]);
draw(leg3c[3],-leg3c[5],leg3c[4]);
draw(leg3c[6],-leg3c[8],leg3c[7]);
draw(leg3c[9],-leg3c[11],leg3c[10]);

// draws leg4
color(GREEN);
linewidth(5);
move(leg4c[0],-leg4c[2],leg4c[1]);
draw(leg4c[3],-leg4c[5],leg4c[4]);
draw(leg4c[6],-leg4c[8],leg4c[7]);
draw(leg4c[9],-leg4c[11],leg4c[10]);

// draws leg5
color(GREEN);
linewidth(5);
 move(leg5c[0],-leg5c[2],leg5c[1]);
draw(leg5c[3],-leg5c[5],leg5c[4]);
draw(leg5c[6],-leg5c[8],leg5c[7]);
draw(leg5c[9],-leg5c[11],leg5c[10]);

// draws leg6
color(GREEN);
linewidth(5);
move(leg6c[0],-leg6c[2],leg6c[1]);
draw(leg6c[3],-leg6c[5],leg6c[4]);
draw(leg6c[6],-leg6c[8],leg6c[7]);
draw(leg6c[9],-leg6c[11],leg6c[10]);

}
```

```
// ***************************************************************
// FILENAME: bot.H
// PURPOSE: defines constants and functions used int bot.C
// NOTE: This is an IRIS 3D program written in C++
// ***************************************************************

#ifndef _BOT_H
#define _BOT_H

// provides constants for menu processing options
#define CAMERA 1
#define ABOVE 2
#define LEG56_VIEW 3
#define LEG1_VIEW 4
#define LEG4_VIEW 5
#define FILEREAD 6
#define RESETFILE 8
#define LEG23_VIEW 9
#define RESET 14
#define EXIT 15

#define NEARCLIPPING 10.0 // planes defined
// #define FARCLIPPING 1023.0
#define FARCLIPPING 2047.0 // changed 14MAY93; C.Schue

long makethemenus();

static float rfx;/* reference point on in the x direction */
static float rfy;/* reference point on in the y direction */
static float rfz;/* reference point on in the z direction */

static float vx; /* view point on in the x direction */
static float vy; /* view point on in the y direction */
static float vz; /* view point on in the z direction */

double delta1,delta2,delta3;
double delaz,delel,delrol,delx,dely,delz;

void processmenuhit(long hititem);

void initialize(); // initializes graphics layout

void loadunit(); // a unit matrix used in rotation/translation

void projectionandviewingmatrix(float vx, float vy, float vz,
        float refx, float refy, float refz);

void buildnonmovingviewingmatrix(float vx, float vy, float vz,
        float refx, float refy, float refz);

void drawaqua(double *, double *, double *, double *,
        double *, double *, double *); // includes all legs and body

#endif

// ***************************************************************************
// FILENAME: Kinematics.C
// PURPOSE: to determine positions(x,y,z) from the H_matrix
//      to read from a file the new link angle changes
//      and update the appropiate leg/link values
// ***************************************************************************

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include "MatrixMy.H"
```

```
#include "AbLeg.H"
#include "AbBody.H"
#include "Link.H"
#include "Link0.H"
#include "Kinematics.H"

#include "arconst.H"

#define GROUNDELEVATION 0.0

// ***************************************************************************
// xyz coordinates are determined from the H_matrix  and
// return the coordinates using the Return_Coordinates structure
// ***************************************************************************
Return_Coordinates FindPositions(AquarobotBody &body,
                                                                    AquaLeg
&leg1, AquaLeg &leg2, AquaLeg &leg3,
                                                                    AquaLeg
&leg4, AquaLeg &leg5, AquaLeg &leg6)
{
 Return_Coordinates *rc;
 rc = new Return_Coordinates;
 // body center coordinates
 rc->bodyc[0] =  body.body_list->val(0,0);
 rc->bodyc[1] =  body.body_list->val(0,1);
 rc->bodyc[2] =  body.body_list->val(0,2);
 // body points to draw
 rc->bodyc[3] = body.body_list->val(1,0);
 rc->bodyc[4] = body.body_list->val(1,1);
 rc->bodyc[5] = body.body_list->val(1,2);
 rc->bodyc[6] = body.body_list->val(2,0);
 rc->bodyc[7] = body.body_list->val(2,1);
 rc->bodyc[8] = body.body_list->val(2,2);
 rc->bodyc[9] = body.body_list->val(3,0);
 rc->bodyc[10] = body.body_list->val(3,1);
 rc->bodyc[11] = body.body_list->val(3,2);
 rc->bodyc[12] = body.body_list->val(4,0);
 rc->bodyc[13] = body.body_list->val(4,1);
 rc->bodyc[14] = body.body_list->val(4,2);
 rc->bodyc[15] = body.body_list->val(5,0);
 rc->bodyc[16] = body.body_list->val(5,1);
 rc->bodyc[17] = body.body_list->val(5,2);
 rc->bodyc[18] = body.body_list->val(6,0);
 rc->bodyc[19] = body.body_list->val(6,1);
 rc->bodyc[20] = body.body_list->val(6,2);

 // prints out body coordinates
 /*
 printf("body center %3f , %3f , %3f \n",rc->bodyc[0],rc->bodyc[1],
 rc->bodyc[2]);
 printf("body pt 1  %3f, %3f, %3f \n",rc->bodyc[3],rc->bodyc[4],
 rc->bodyc[5]);
 printf("body pt 2 %3f, %3f,  %3f \n",rc->bodyc[6],rc->bodyc[7],
 rc->bodyc[8]);
 printf("body pt 3 %3f, %3f, %3f \n",rc->bodyc[9],rc->bodyc[10],
 rc->bodyc[11]);
 printf("body pt 4 %3f, %3f, %3f \n",rc->bodyc[12],rc->bodyc[13],
 rc->bodyc[14]);
 printf("body pt 5 %3f, %3f, %3f \n",rc->bodyc[15],rc->bodyc[16],
 rc->bodyc[17]);
 printf("body pt 6 %3f, %3f, %3f \n\n",rc->bodyc[18],rc->bodyc[19],
 rc->bodyc[20]);
 */

 // joint one leg coordinates: [0]=x [1]=y [2]=z
 rc->leg1c[0] = leg1.link0->H_matrix->val(0,3);
 rc->leg2c[0] = leg2.link0->H_matrix->val(0,3);
```

```
rc->leg3c[0] = leg3.link0->H_matrix->val(0,3);
rc->leg4c[0] = leg4.link0->H_matrix->val(0,3);
rc->leg5c[0] = leg5.link0->H_matrix->val(0,3);
rc->leg6c[0] = leg6.link0->H_matrix->val(0,3);
rc->leg1c[1] = leg1.link0->H_matrix->val(1,3);
rc->leg2c[1] = leg2.link0->H_matrix->val(1,3);
rc->leg3c[1] = leg3.link0->H_matrix->val(1,3);
rc->leg4c[1] = leg4.link0->H_matrix->val(1,3);
rc->leg5c[1] = leg5.link0->H_matrix->val(1,3);
rc->leg6c[1] = leg6.link0->H_matrix->val(1,3);

rc->leg1c[2] = leg1.link0->H_matrix->val(2,3);
rc->leg2c[2] = leg2.link0->H_matrix->val(2,3);
rc->leg3c[2] = leg3.link0->H_matrix->val(2,3);
rc->leg4c[2] = leg4.link0->H_matrix->val(2,3);
rc->leg5c[2] = leg5.link0->H_matrix->val(2,3);
rc->leg6c[2] = leg6.link0->H_matrix->val(2,3);

// inboard joint angle info. HIP joint angle
//   [][] = [leg][joint]
rc->inbd_joint_angle[0][1] = leg1.link0->GetInboardJointAngle();
rc->inbd_joint_angle[1][1] = leg2.link0->GetInboardJointAngle();
rc->inbd_joint_angle[2][1] = leg3.link0->GetInboardJointAngle();
rc->inbd_joint_angle[3][1] = leg4.link0->GetInboardJointAngle();
rc->inbd_joint_angle[4][1] = leg5.link0->GetInboardJointAngle();
rc->inbd_joint_angle[5][1] = leg6.link0->GetInboardJointAngle();

// joint 2 x,y,z coordinates [3]=x [4]=y [5]=z
rc->leg1c[3] = leg1.link1->H_matrix->val(0,3);
rc->leg2c[3] = leg2.link1->H_matrix->val(0,3);
rc->leg3c[3] = leg3.link1->H_matrix->val(0,3);
rc->leg4c[3] = leg4.link1->H_matrix->val(0,3);
rc->leg5c[3] = leg5.link1->H_matrix->val(0,3);
rc->leg6c[3] = leg6.link1->H_matrix->val(0,3);
rc->leg1c[4] = leg1.link1->H_matrix->val(1,3);
rc->leg2c[4] = leg2.link1->H_matrix->val(1,3);
rc->leg3c[4] = leg3.link1->H_matrix->val(1,3);
rc->leg4c[4] = leg4.link1->H_matrix->val(1,3);
rc->leg5c[4] = leg5.link1->H_matrix->val(1,3);
rc->leg6c[4] = leg6.link1->H_matrix->val(1,3);

rc->leg1c[5] = leg1.link1->H_matrix->val(2,3);
rc->leg2c[5] = leg2.link1->H_matrix->val(2,3);
rc->leg3c[5] = leg3.link1->H_matrix->val(2,3);
rc->leg4c[5] = leg4.link1->H_matrix->val(2,3);
rc->leg5c[5] = leg5.link1->H_matrix->val(2,3);
rc->leg6c[5] = leg6.link1->H_matrix->val(2,3);

// joint 2 motion_limit_flag
rc->motion_limit_flag[0] = leg1.link1->GetMotionLimitFlag();
rc->motion_limit_flag[3] = leg2.link1->GetMotionLimitFlag();
rc->motion_limit_flag[6] = leg3.link1->GetMotionLimitFlag();
rc->motion_limit_flag[9] = leg4.link1->GetMotionLimitFlag();
rc->motion_limit_flag[12] = leg5.link1->GetMotionLimitFlag();
rc->motion_limit_flag[15] = leg6.link1->GetMotionLimitFlag();

// inboard joint angle info. KNEE1 joint angle
//   [][] = [leg][joint]
rc->inbd_joint_angle[0][2] = leg1.link1->GetInboardJointAngle();
rc->inbd_joint_angle[1][2] = leg2.link1->GetInboardJointAngle();
rc->inbd_joint_angle[2][2] = leg3.link1->GetInboardJointAngle();
rc->inbd_joint_angle[3][2] = leg4.link1->GetInboardJointAngle();
rc->inbd_joint_angle[4][2] = leg5.link1->GetInboardJointAngle();
rc->inbd_joint_angle[5][2] = leg6.link1->GetInboardJointAngle();

// joint 3 xyz coordinates [6]=x [7]=y [8]=z
rc->leg1c[6] = leg1.link2->H_matrix->val(0,3);
```

```
rc->leg2c[6] = leg2.link2->H_matrix->val(0,3);
rc->leg3c[6] = leg3.link2->H_matrix->val(0,3);
rc->leg4c[6] = leg4.link2->H_matrix->val(0,3);
rc->leg5c[6] = leg5.link2->H_matrix->val(0,3);
rc->leg6c[6] = leg6.link2->H_matrix->val(0,3);

rc->leg1c[7] = leg1.link2->H_matrix->val(1,3);
rc->leg2c[7] = leg2.link2->H_matrix->val(1,3);
rc->leg3c[7] = leg3.link2->H_matrix->val(1,3);
rc->leg4c[7] = leg4.link2->H_matrix->val(1,3);
rc->leg5c[7] = leg5.link2->H_matrix->val(1,3);
rc->leg6c[7] = leg6.link2->H_matrix->val(1,3);

rc->leg1c[8] = leg1.link2->H_matrix->val(2,3);
rc->leg2c[8] = leg2.link2->H_matrix->val(2,3);
rc->leg3c[8] = leg3.link2->H_matrix->val(2,3);
rc->leg4c[8] = leg4.link2->H_matrix->val(2,3);
rc->leg5c[8] = leg5.link2->H_matrix->val(2,3);
rc->leg6c[8] = leg6.link2->H_matrix->val(2,3);

// joint 3 motion_limit_flag
rc->motion_limit_flag[1] = leg1.link2->GetMotionLimitFlag();
rc->motion_limit_flag[4] = leg2.link2->GetMotionLimitFlag();
rc->motion_limit_flag[7] = leg3.link2->GetMotionLimitFlag();
rc->motion_limit_flag[10] = leg4.link2->GetMotionLimitFlag();
rc->motion_limit_flag[13] = leg5.link2->GetMotionLimitFlag();
rc->motion_limit_flag[16] = leg6.link2->GetMotionLimitFlag();

// inboard joint angle info. KNEE2 joint angle
//   [][] = [leg][joint]
rc->inbd_joint_angle[0][3] = leg1.link2->GetInboardJointAngle();
rc->inbd_joint_angle[1][3] = leg2.link2->GetInboardJointAngle();
rc->inbd_joint_angle[2][3] = leg3.link2->GetInboardJointAngle();
rc->inbd_joint_angle[3][3] = leg4.link2->GetInboardJointAngle();
rc->inbd_joint_angle[4][3] = leg5.link2->GetInboardJointAngle();
rc->inbd_joint_angle[5][3] = leg6.link2->GetInboardJointAngle();

// joint 4 xyz coordinates [9]=x [10]=y [11]=z
rc->leg1c[9] = leg1.link3->H_matrix->val(0,3);
rc->leg2c[9] = leg2.link3->H_matrix->val(0,3);
rc->leg3c[9] = leg3.link3->H_matrix->val(0,3);
rc->leg4c[9] = leg4.link3->H_matrix->val(0,3);
rc->leg5c[9] = leg5.link3->H_matrix->val(0,3);
rc->leg6c[9] = leg6.link3->H_matrix->val(0,3);

rc->leg1c[10] = leg1.link3->H_matrix->val(1,3);
rc->leg2c[10] = leg2.link3->H_matrix->val(1,3);
rc->leg3c[10] = leg3.link3->H_matrix->val(1,3);
rc->leg4c[10] = leg4.link3->H_matrix->val(1,3);
rc->leg5c[10] = leg5.link3->H_matrix->val(1,3);
rc->leg6c[10] = leg6.link3->H_matrix->val(1,3);

rc->leg1c[11] = leg1.link3->H_matrix->val(2,3);
rc->leg2c[11] = leg2.link3->H_matrix->val(2,3);
rc->leg3c[11] = leg3.link3->H_matrix->val(2,3);
rc->leg4c[11] = leg4.link3->H_matrix->val(2,3);
rc->leg5c[11] = leg5.link3->H_matrix->val(2,3);
rc->leg6c[11] = leg6.link3->H_matrix->val(2,3);

// joint 4 motion_limit_flag
rc->motion_limit_flag[2] = leg1.link3->GetMotionLimitFlag();
rc->motion_limit_flag[5] = leg2.link3->GetMotionLimitFlag();
rc->motion_limit_flag[8] = leg3.link3->GetMotionLimitFlag();
rc->motion_limit_flag[11] = leg4.link3->GetMotionLimitFlag();
rc->motion_limit_flag[14] = leg5.link3->GetMotionLimitFlag();
rc->motion_limit_flag[17] = leg6.link3->GetMotionLimitFlag();
```

```
// inboard joint angle info. FOOT joint angle
//   [][] = [leg][joint]
rc->inbd_joint_angle[0][4] = leg1.link3->GetInboardJointAngle();
rc->inbd_joint_angle[1][4] = leg2.link3->GetInboardJointAngle();
rc->inbd_joint_angle[2][4] = leg3.link3->GetInboardJointAngle();
rc->inbd_joint_angle[3][4] = leg4.link3->GetInboardJointAngle();
rc->inbd_joint_angle[4][4] = leg5.link3->GetInboardJointAngle();
rc->inbd_joint_angle[5][4] = leg6.link3->GetInboardJointAngle();

// test for supporting legs and adjusting leg_support_flag
if (fabs(rc->leg1c[11]) >= GROUNDELEVATION) leg1.SetLegSupportFlag(1);
else leg1.SetLegSupportFlag(0);
if (fabs(rc->leg2c[11]) >= GROUNDELEVATION) leg2.SetLegSupportFlag(1);
else leg2.SetLegSupportFlag(0);
if (fabs(rc->leg3c[11]) >= GROUNDELEVATION) leg3.SetLegSupportFlag(1);
else leg3.SetLegSupportFlag(0);
if (fabs(rc->leg4c[11]) >= GROUNDELEVATION) leg4.SetLegSupportFlag(1);
else leg4.SetLegSupportFlag(0);
if (fabs(rc->leg5c[11]) >= GROUNDELEVATION) leg5.SetLegSupportFlag(1);
else leg5.SetLegSupportFlag(0);
if (fabs(rc->leg6c[11]) >= GROUNDELEVATION) leg6.SetLegSupportFlag(1);
else leg6.SetLegSupportFlag(0);

//  places leg_support_flag into rc
rc->leg_support_flag[0] = leg1.GetLegSupportFlag();
rc->leg_support_flag[1] = leg2.GetLegSupportFlag();
rc->leg_support_flag[2] = leg3.GetLegSupportFlag();
rc->leg_support_flag[3] = leg4.GetLegSupportFlag();
rc->leg_support_flag[4] = leg5.GetLegSupportFlag();
rc->leg_support_flag[5] = leg6.GetLegSupportFlag();

// prints body and leg xyz coordinates
/*
int row, col;
printf("leg1                    leg2\n");
for( row = 1; row<5; row++){
        for(col = 3; col>0; col--)
                printf("%6.4f  ",rc->leg1c[3 * row - col]);
        printf("   ");
        for(col = 3; col>0; col--)
          printf("%6.4f  ",rc->leg2c[3 * row - col]);
        printf("\n");
}
printf("\n");
printf("leg3                    leg4\n");
for(row = 1; row<5; row++){
        for(col = 3;col>0; col--)
                printf("%6.4f ",rc->leg3c[3 * row - col]);
        printf("   ");
        for( col = 3; col>0; col--)
                printf("%6.4f ",rc->leg4c[3 * row - col]);
        printf("\n");
}
printf("\n");
printf("leg5                    leg6\n");
for(row = 1; row<5; row++){
        for(col = 3; col>0; col--)
                printf("%6.4f ",rc->leg5c[3 * row - col]);
        printf("   ");
        for(col = 3; col>0; col--)
                printf("%6.4f ",rc->leg6c[3 * row - col]);
        printf("\n");
}
printf("\n");
*/
return *rc;
}
```

```
// ************************************************************************
// FUNCTION: File_Use
// PURPOSE: reads desired leg changes from a file
// INPUT: reads from file: leg#, delta1, delta2, delta3
// OUTPUT: calculates new leg/link coordinates (internal)
// ************************************************************************
Passing_Items File_Use(FILE *ifp, AquarobotBody &body,
                                                    AquaLeg &leg1, AquaLeg &leg2, AquaLeg &leg3,
                                                    AquaLeg &leg4, AquaLeg &leg5, AquaLeg &leg6)
{
 Passing_Items *pass;
 pass = new Passing_Items;

 fscanf(ifp,"%d %lf %lf %lf %lf %lf %lf",&pass->body,&pass->del1,
                        &pass->del2,&pass->del3,&pass->del4,&pass->del5,&pass->del6);

 if (pass->legnum < 9) {
          body.MoveIncremental(pass->del1, pass->del2, pass->del3,
                                                              pass->del4, pass->del5, pass->del6);
          fscanf(ifp,"%d %lf %lf %lf",
                        &pass->legnum, &pass->del1, &pass->del2, &pass->del3);
          leg1.MoveIncremental(body,pass->del1,pass->del2,pass->del3);
          fscanf(ifp,"%d %lf %lf %lf",
                        &pass->legnum, &pass->del1, &pass->del2, &pass->del3);
          leg2.MoveIncremental(body,pass->del1,pass->del2,pass->del3);
          fscanf(ifp,"%d %lf %lf %lf",
                        &pass->legnum, &pass->del1, &pass->del2, &pass->del3);
          leg3.MoveIncremental(body,pass->del1,pass->del2,pass->del3);
          fscanf(ifp,"%d %lf %lf %lf",
                        &pass->legnum, &pass->del1, &pass->del2, &pass->del3);
          leg4.MoveIncremental(body,pass->del1,pass->del2,pass->del3);
          fscanf(ifp,"%d %lf %lf %lf",
                        &pass->legnum, &pass->del1, &pass->del2, &pass->del3);
          leg5.MoveIncremental(body,pass->del1,pass->del2,pass->del3);
          fscanf(ifp,"%d %lf %lf %lf",
                        &pass->legnum, &pass->del1, &pass->del2, &pass->del3);
          leg6.MoveIncremental(body,pass->del1,pass->del2,pass->del3);
          fscanf(ifp,"%d %lf %lf %lf",
                        &pass->legnum, &pass->del1, &pass->del2, &pass->del3);

          if (pass->legnum == 0) {
            pass->del1 = 0.0;
            pass->del2 = 0.0;
            pass->del3 = 0.0;
          } ;
 };
 return *pass;
}

// ************************************************************************
// FILENAME:  TransferToGait
// PURPOSE: places the body center and leg coordinates in a
//        Next_Motion structure for gait algorithm use
// ************************************************************************
Next_Motion TransferToGait(Return_Coordinates &coord, AquarobotBody &body)
{
 Next_Motion *temp;
 temp = new Next_Motion;

 temp->body_center_coord[3] = coord.bodyc[0];
 temp->body_center_coord[4] = coord.bodyc[1];
 temp->body_center_coord[5] = coord.bodyc[2];

 temp->foot_1_coord[0] = coord.leg1c[9];
 temp->foot_1_coord[1] = coord.leg1c[10];
 temp->foot_1_coord[2] = coord.leg1c[11];
```

```
temp->foot_2_coord[0] = coord.leg2c[9];
temp->foot_2_coord[1] = coord.leg2c[10];
temp->foot_2_coord[2] = coord.leg2c[11];

temp->foot_3_coord[0] = coord.leg3c[9];
temp->foot_3_coord[1] = coord.leg3c[10];
temp->foot_3_coord[2] = coord.leg3c[11];

temp->foot_4_coord[0] = coord.leg4c[9];
temp->foot_4_coord[1] = coord.leg4c[10];
temp->foot_4_coord[2] = coord.leg4c[11];

temp->foot_5_coord[0] = coord.leg5c[9];
temp->foot_5_coord[1] = coord.leg5c[10];
temp->foot_5_coord[2] = coord.leg5c[11];

temp->foot_6_coord[0] = coord.leg6c[9];
temp->foot_6_coord[1] = coord.leg6c[10];
temp->foot_6_coord[2] = coord.leg6c[11];

// get joint angle values from algorithm module
for (int i=0; i<6; i++) {
        for (int j=0; j<4; j++) // HIP --> FOOT
                temp->inbd_joint_angle[i][j] = coord.inbd_joint_angle[i][j];
};

// current body elevation
temp->body_center_coord[1] = -1. * body.H_matrix->val(2,0);

// current body azimuth
temp->body_center_coord[0] =asin(body.H_matrix->val(1,0) /
                                                        cos(temp->body_center_coord[1]));

// current body roll
temp->body_center_coord[2]=asin(body.H_matrix->val(2,1) /
                                                        cos(temp->body_center_coord[1]));

// joint_limit_flag
temp->joint_limit_flag[0] = coord.motion_limit_flag[0];
temp->joint_limit_flag[1] = coord.motion_limit_flag[1];
temp->joint_limit_flag[2] = coord.motion_limit_flag[2];
temp->joint_limit_flag[3] = coord.motion_limit_flag[3];
temp->joint_limit_flag[4] = coord.motion_limit_flag[4];
temp->joint_limit_flag[5] = coord.motion_limit_flag[5];
temp->joint_limit_flag[6] = coord.motion_limit_flag[6];
temp->joint_limit_flag[7] = coord.motion_limit_flag[7];
temp->joint_limit_flag[8] = coord.motion_limit_flag[8];
temp->joint_limit_flag[9] = coord.motion_limit_flag[9];
temp->joint_limit_flag[10] = coord.motion_limit_flag[10];
temp->joint_limit_flag[11] = coord.motion_limit_flag[11];
temp->joint_limit_flag[12] = coord.motion_limit_flag[12];
temp->joint_limit_flag[13] = coord.motion_limit_flag[13];
temp->joint_limit_flag[14] = coord.motion_limit_flag[14];
temp->joint_limit_flag[15] = coord.motion_limit_flag[15];
temp->joint_limit_flag[16] = coord.motion_limit_flag[16];

// leg_contact_flag
temp->leg_contact_flag[0] = coord.leg_support_flag[0];
temp->leg_contact_flag[1] = coord.leg_support_flag[1];
temp->leg_contact_flag[2] = coord.leg_support_flag[2];
temp->leg_contact_flag[3] = coord.leg_support_flag[3];
temp->leg_contact_flag[4] = coord.leg_support_flag[4];
temp->leg_contact_flag[5] = coord.leg_support_flag[5];
temp->leg_contact_flag[6] = coord.leg_support_flag[6];

return *temp;
```

```
}

#ifndef KINEMATICS_H
#define KINEMATICS_H

// ***************************************************************************
// FILENAME: Kinematics.H
// ***************************************************************************

#include "arconst.H"

#include "MatrixMy.H"
#include "AbLeg.H"
#include "AbBody.H"

// structure designed to receive file input
struct Passing_Items {
  int legnum;
  int body;
  double del1;
  double del2;
  double del3;
  double del4;
  double del5;
  double del6;
};

// structure designed to return the xyz coordinates of the robot
struct Return_Coordinates {
  double bodyc[21];
  double leg1c[12];
  double leg2c[12];
  double leg3c[12];
  double leg4c[12];
  double leg5c[12];
  double leg6c[12];
  int motion_limit_flag[18];
  int leg_support_flag[6];
  double inbd_joint_angle[6][5];
};

// structure to recieve next robot motion from gait planning algorithms
struct Next_Motion {
 public:
 double bodymotion[6];
 double leg1motion[3];
 double leg2motion[3];
 double leg3motion[3];
 double leg4motion[3];
 double leg5motion[3];
 double leg6motion[3];
 int   leg_contact_flag[6];
 int   joint_limit_flag[18];
 double inbd_joint_angle[LEG][5];
 double foot_1_coord[3];
 double foot_2_coord[3];
 double foot_3_coord[3];
 double foot_4_coord[3];
 double foot_5_coord[3];
 double foot_6_coord[3];
 double body_center_coord[6];

 // constructor
 Next_Motion() {

        bodymotion[0] = 0.0; bodymotion[1] = 0.0; bodymotion[2] = 0.0;
        bodymotion[3] = 0.0; bodymotion[4] = 0.0; bodymotion[5] = 0.0;
```

```
leg_contact_flag[LEG1] = 1; leg_contact_flag[LEG2] = 1;
leg_contact_flag[LEG3] = 1; leg_contact_flag[LEG4] = 1;
leg_contact_flag[LEG5] = 1; leg_contact_flag[LEG6] = 1;

leg1motion[X] = 0.0; leg1motion[Y] = 0.0; leg1motion[Z] = 0.0;
leg2motion[X] = 0.0; leg2motion[Y] = 0.0; leg2motion[Z] = 0.0;
leg3motion[X] = 0.0; leg3motion[Y] = 0.0; leg3motion[Z] = 0.0;
leg4motion[X] = 0.0; leg4motion[Y] = 0.0; leg4motion[Z] = 0.0;
leg5motion[X] = 0.0; leg5motion[Y] = 0.0; leg5motion[Z] = 0.0;
leg6motion[X] = 0.0; leg6motion[Y] = 0.0; leg6motion[Z] = 0.0;

joint_limit_flag[0] = 0; joint_limit_flag[1] = 0;
joint_limit_flag[2] = 0; joint_limit_flag[3] = 0;
joint_limit_flag[4] = 0; joint_limit_flag[5] = 0;
joint_limit_flag[6] = 0; joint_limit_flag[7] = 0;
joint_limit_flag[8] = 0; joint_limit_flag[9] = 0;
joint_limit_flag[10] = 0; joint_limit_flag[11] = 0;
joint_limit_flag[12] = 0; joint_limit_flag[13] = 0;
joint_limit_flag[14] = 0; joint_limit_flag[15] = 0;
joint_limit_flag[16] = 0; joint_limit_flag[17] = 0;

inbd_joint_angle[LEG1][CB]  = (double)LEG1*60.0;
inbd_joint_angle[LEG1][HIP]  = StartTheta1;
inbd_joint_angle[LEG1][KNEE1]= StartTheta2;
inbd_joint_angle[LEG1][KNEE2]= StartTheta3;
inbd_joint_angle[LEG1][FOOT] = 0.0;

inbd_joint_angle[LEG2][CB]  = (double)LEG2*60.0;
inbd_joint_angle[LEG2][HIP]  = StartTheta1;
inbd_joint_angle[LEG2][KNEE1]= StartTheta2;
inbd_joint_angle[LEG2][KNEE2]= StartTheta3;
inbd_joint_angle[LEG2][FOOT] = 0.0;

inbd_joint_angle[LEG3][CB]  = (double)LEG3*60.0;
inbd_joint_angle[LEG3][HIP]  = StartTheta1;
inbd_joint_angle[LEG3][KNEE1]= StartTheta2;
inbd_joint_angle[LEG3][KNEE2]= StartTheta3;
inbd_joint_angle[LEG3][FOOT] = 0.0;

inbd_joint_angle[LEG4][CB]  = (double)LEG4*60.0;
inbd_joint_angle[LEG4][HIP]  = StartTheta1;
inbd_joint_angle[LEG4][KNEE1]= StartTheta2;
inbd_joint_angle[LEG4][KNEE2]= StartTheta3;
inbd_joint_angle[LEG4][FOOT] = 0.0;

inbd_joint_angle[LEG5][CB]  = (double)LEG5*60.0;
inbd_joint_angle[LEG5][HIP]  = StartTheta1;
inbd_joint_angle[LEG5][KNEE1]= StartTheta2;
inbd_joint_angle[LEG5][KNEE2]= StartTheta3;
inbd_joint_angle[LEG5][FOOT] = 0.0;

inbd_joint_angle[LEG6][CB]  = (double)LEG6*60.0;
inbd_joint_angle[LEG6][HIP]  = StartTheta1;
inbd_joint_angle[LEG6][KNEE1]= StartTheta2;
inbd_joint_angle[LEG6][KNEE2]= StartTheta3;
inbd_joint_angle[LEG6][FOOT] = 0.0;

/* respect to WORLD Coordinates */
foot_1_coord[XW]= 98.02; foot_1_coord[YW]=  0.0;    foot_1_coord[ZW]=0.0;
foot_2_coord[XW]= 49.01; foot_2_coord[YW]= 84.88781; foot_2_coord[ZW]=0.0;
foot_3_coord[XW]=-49.01; foot_3_coord[YW]= 84.88781; foot_3_coord[ZW]=0.0;
foot_4_coord[XW]=-98.02; foot_4_coord[YW]=  0.0;    foot_4_coord[ZW]=0.0;
foot_5_coord[XW]=-49.01; foot_5_coord[YW]=-84.88781; foot_5_coord[ZW]=0.0;
foot_6_coord[XW]= 49.01, foot_6_coord[YW]=-84.88781; foot_6_coord[ZW]=0.0;

body_center_coord[0] = 0.0;
```

```
   body_center_coord[1] = 0.0;
   body_center_coord[2] = 0.0;
   body_center_coord[3] = 0.0;
   body_center_coord[4] = 0.0;
   body_center_coord[5] = -70.71;
  };
};

// structure
struct WalkParameter {
 public:
  int phase;
  double stride;
  double direction;
  double tripodsize;
  double footheight;
  double bodyspeed;

  WalkParameter() {
            phase = 0;
            stride = DefaultSTRIDE;
            direction = 0.0;
            tripodsize = 98.02;
            footheight = DefaultFOOTheight;
            bodyspeed = (DefaultSTRIDE/2.0)/FINE;
  };
};

// structure
struct Flag {
 public:
  int phaseEnd;
  int goal;

  // constructor
  Flag() {
   phaseEnd = 0;
   goal = 0;
  };
};

extern
Return_Coordinates FindPositions(AquarobotBody &,
                         AquaLeg &, AquaLeg &, AquaLeg &,
                         AquaLeg &, AquaLeg &, AquaLeg &);

extern
Passing_Items File_Use(FILE *, AquarobotBody &,
                        AquaLeg &, AquaLeg &, AquaLeg &,
                         AquaLeg &, AquaLeg &, AquaLeg &);

extern
Next_Motion TransferToGait(Return_Coordinates &, AquarobotBody &);

#endif

// ********************************************************************
// FILENAME: Link0.C
// PURPOSE: Declarations for class Link0
// ********************************************************************

#include "Link0.H"

Link0::Link0() : Link ( 0, 37.5, 0.0, 0.0, 0.0, -1.0,
              -360.0,360.0)
{
 node_list = new matrix(4,4,0.0);
```

```
  node_list->val(0,3) =1.; node_list->val(1,3) =1.;
  node_list->val(2,0) =37.5; node_list->val(2,3) = 1.;

  T_matrix = new matrix(4,4,0.0);

}

// ********************************************************************
// FILENAME: Link0.H
// PURPOSE: Declarations for class Link0
// ********************************************************************

#ifndef H_LINK0
#define H_LINK0

#include "Link.H"

class Link0 : public Link
{
private:
public:
  Link0();

};

#endif

// ********************************************************************
// FILENAME: Link1.C
// ********************************************************************

#include "Link1.H"

// Link1::Link1() : Link ( 0, 20.0, -90.0, 66.4, 0.0, 0,-106.6,73.4)
// Changed by Kenji Suzuki & Chuck Schue
Link1::Link1() : Link ( 0, 20.0, -90.0, 35.86, 0.0, 0,-106.6,73.4)
{
  node_list = new matrix(4,4,0.0);
  node_list->val(0,3) =1.;
  node_list->val(1,3) =1.;
  node_list->val(2,0) = 20.0;
  node_list->val(2,3) = 1.;

  T_matrix = new matrix(4,4,0.0);

}

// ********************************************************************
// FILENAME: Link1.H
// PURPOSE: Declarations for class Link0
// ********************************************************************

#ifndef H_LINK1
#define H_LINK1

#include "Link.H"

class Link1 : public Link
{

private:

public:
        Link1();
};

#endif
```

```
// ****************************************************************
// FILENAME: Link2.C
// PURPOSE: Declarations for class Link0
// ****************************************************************

#include "Link2.H"

// Link2::Link2() : Link ( 0, 50.0, 0.0, -156.4, 0.0, 1.0, -156.4, 23.6)
// Changed by Kenji Suzuki & Chuck Schue
Link2::Link2() : Link ( 0, 50.0, 0.0, -125.86, 0.0, 1.0, -156.4, 23.6)
{
 node_list = new matrix(4,4,0.0);
 node_list->val(0,3) =1.; node_list->val(1,3) =1.;
 node_list->val(2,0) =50.; node_list->val(2,3) = 1.;

 T_matrix = new matrix(4,4,0.0);

}

// ****************************************************************
// FILENAME: Link2.H
// PURPOSE: Declarations for class Link0
// ****************************************************************

#ifndef H_LINK2
#define H_LINK2

#include "Link.H"

class Link2 : public Link
{

private:

public:
        Link2();
};
#endif

// ****************************************************************
// FILENAME: Link3.C
// PURPOSE: Declarations for class Link0
// ****************************************************************

#include "Link3.H"

Link3::Link3() : Link ( 0, 100.0, 0.0,0.0, 0.0, 2.0, -360.0,360.0)
{
 node_list = new matrix(4,4,0.0);
 node_list->val(0,3) =1.; node_list->val(1,3) =1.;
 node_list->val(2,0) =100.; node_list->val(2,3) = 1.;

 T_matrix = new matrix(4,4,0.0);

}

// ****************************************************************
// FILENAME: Link3.H
// PURPOSE: Declarations for class Link0
// ****************************************************************

#ifndef H_LINK3
#define H_LINK3

#include "Link.H"
```

```
class Link3 : public Link
{

private:

public:
        Link3();
};

#endif

// ********************************************************
// FILENAME: Link.C
// PURPOSE: Implementation of class Link
// CONTAINS: UpdateAMatrix ()
//       Rotate (double angle)
//       RotateLink (double angle)
// ********************************************************

#include "Link.H"

const int True = 1;
const int False = 0;

// ********************************************************
// FUNCTION: Link
// PURPOSE:  Constructor for Link
// ********************************************************
Link::Link ( int mlf, double ll, double ta, double ija, double ijd, double il,
    double min_ja, double max_ja )
{
 motion_limit_flag = mlf;
 link_length = ll;
 twist_angle = ta;
 inboard_joint_angle = ija;
 inboard_joint_displacement = ijd;
 inboard_link = il;
 min_joint_angle = min_ja;
 max_joint_angle = max_ja;
 H_matrix = new matrix(4,4,0.0);

 H_matrix->UpdateTMatrix(ija,ta,ll,ijd);
}

// ********************************************************
// FUNCTION: ~Link
// PURPOSE: destructor for Link class
// ********************************************************
Link::~Link()
{
  delete T_matrix;
  delete node_list;
}

// ********************************************************
// FUNCTION: Rotate
// PURPOSE: rotates a Link by changing the T Matrix
//      by the inboard joint angle desired
// ********************************************************
void Link::Rotate (matrix *mat, double angle)
{
 SetInboardJointAngle(angle);

 T_matrix->UpdateTMatrix(GetInboardJointAngle(),GetTwistAngle(),
          GetLinkLength(),GetInboardJointDisplacement());

 // the "mat" is the inboard link's A matrix (or the body's
```

```
 // A matrix for the inboard joint
 *H_matrix = *mat * *T_matrix;

}

// ***********************************************************
// FUNCTION: RotateLink
// PURPOSE: determines if the rotation is within physical
//    joint constraints. If outside the workspace the min
//    or max limit applicable is used.
//        this function calls the Rotate function
// ***********************************************************
void Link::RotateLink(matrix *mat, double angle)
{
 double tester;
 tester = GetMinJointAngle();
 if (angle < tester) {
            angle = tester;
            SetMotionLimitFlag(1);
 }
 tester = GetMaxJointAngle();
 if (angle > tester) {
            angle = tester;
            SetMotionLimitFlag(1);
 }
 Rotate(mat, angle);
}

// ***********************************************************
// FILENAME: Link.H
// PURPOSE: Declarations for class Link
// COMMENTS: Definition of Link class
// ***********************************************************

#ifndef H_LINK
#define H_LINK

#include <stdio.h>
#include <math.h>
#include "AbRigid.H"
#include "MatrixMy.H"

class Link: public RigidBody
{
private:
  int motion_limit_flag;
  double link_length;
  double twist_angle;
  double inboard_joint_angle;
  double inboard_joint_displacement;
  double inboard_link;
  double min_joint_angle; // rotary link
  double max_joint_angle; // rotary link

public:
  void RotateLink(matrix*, double);

  matrix *T_matrix, *node_list, *H_matrix;

  Link ( int mlf, double ll, double ta, double ija, double ijd, double il,
                                    double min_ja, double max_ja );
  ~Link();

  int GetMotionLimitFlag() {return motion_limit_flag;}
  double GetLinkLength() {return link_length;}
  double GetTwistAngle() {return twist_angle;}
  double GetInboardJointAngle() {return inboard_joint_angle;}
```

```
  double GetInboardJointDisplacement() {return inboard_joint_displacement;}
  double GetInboardLink() {return inboard_link;}
  double GetTMatrix(int row, int col) {return T_matrix->val(row,col);}
  double GetMinJointAngle() {return min_joint_angle;}
  double GetMaxJointAngle() {return max_joint_angle;}

  void SetMotionLimitFlag(int a) {motion_limit_flag = a;}
  void SetLinkLength(double a) {link_length = a;}
  void SetTwistAngle(double a) {twist_angle = a;}
  void SetInboardJointAngle(double a) {inboard_joint_angle = a;}
  void SetInboardJointDisplacement(double a) {inboard_joint_displacement =a;}
  void SetInboardLink(double a) {inboard_link = a;}
  void SetTMatrix(int row,int col, double a)
            {T_matrix->val(row,col) = a;}
  void SetMinJointAngle(double a) {min_joint_angle = a;}
  void SetMaxJointAngle(double a) {max_joint_angle = a;}

  void Rotate(matrix*, double);
};

#endif

// ************************************************
// FILENAME: MatrixMy.C
// PURPOSE: Implementation of MatrixMy class
// CONTAINS:
// ************************************************************

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "MatrixMy.H"
#include "AbLeg.H"

matrix::matrix()
{
  p = new matrep;
  p->r = 4;
  p->c = 4;
  p->m = new double *[4];
  int x;
  for (x =0; x<4; x++)
            p->m[x] = new double[4];
  p->n = 1;
  int j;
  for (int i=0; i<4; i++)
            for (j=0; j<4; j++)
            p->m[i][j] = 0.0;
}

matrix::matrix(int rows = 1, int col = 1, double initval = 0.)
{
  p = new matrep;
  p->r = rows;
  p->c = col;
  p->m = new double *[rows];
  int x;
  for (x =0; x<rows; x++)
            p->m[x] = new double[col];
  p->n = 1;
  int j;
  for (int i=0; i<rows; i++)
            for (j=0; j<col; j++)
            p->m[i][j] = initval;
}
```

```
matrix::matrix(const matrix& x)
{
 x.p->n++;
 p = x.p;
}

matrix matrix::operator=(const matrix& rval)
{
 if (--p->n == 0) {
  for (int x=0; x<rows(); x++)
                        delete p->m[x];
  delete p->m;
  delete p;
 }
 rval.p->n++;
 p = rval.p;
 return *this;
}


matrix::~matrix()
{
 if (--p->n == 0) {
            for (int x=0; x<rows(); x++)
                        delete p->m[x];
            delete p->m;
            delete p;
 }
}

double & matrix::val(int row, int col) const
{
 return (p->m[row][col]);
}

matrix matrix::operator*(const matrix& arg)
{
 matrix result(rows(),arg.cols(),0.0);
 for (int row=0; row<rows();row++) {
            int col;
            for (col=0; col<arg.cols(); col++) {
   double sum=0.0;
   for (int i=0; i<cols(); i++)
    sum += p->m[row][i] * arg.val(i,col);
   result.val(row,col) = sum;
            }
 }
 return result;
}

matrix matrix::operator*(double a)
{
 matrix result(rows(),cols(),0.0);
 for (int i=0; i<rows(); i++) {
            for (int j=0; j<cols(); j++) {
                        double ans;
                        ans = result.val(i,j) * a;
                        result.val(i,j) = ans;
            }
 }
 return result;
}

matrix matrix::operator+(const matrix& arg)
{
 matrix sum(rows(),cols(),0.0);
```

```
 for (int i=0; i<rows(); i++) {
            int j;
            for (j=0; j<cols(); j++)
                  sum.p->m[i][j] = p->m[i][j] + arg.val(i,j);
 }
 return sum;
}

void matrix::print()
{
 for (int row=0; row<rows(); row++) {
            int col;
            for (col=0; col<cols(); col++)
                  printf("%6.6f ", p->m[row][col]);
            printf("\n");
 }

}

matrix & matrix::HomogeneousTransform(double azimuth, double elevation,
                        double roll,
                        double x, double y, double z)
{
 double spsi = sin(azimuth);
 double cpsi = cos(azimuth);
 double sth = sin(elevation);
 double cth = cos(elevation);
 double sphi = sin(roll);
 double cphi = cos(roll);
 val(0,0) = (cpsi * cth);
 val(0,1) = ((cpsi * sth * sphi) - (spsi * cphi));
 val(0,2) = ((cpsi * sth * cphi) + (spsi * sphi));
 val(0,3) = x;
 val(1,0) = (spsi * cth);
 val(1,1) = ((cpsi * cphi) + (spsi * sth * sphi));
 val(1,2) = ((spsi * sth * cphi) - (cpsi * sphi));
 val(1,3) = y;
 val(2,0) = (-sth);
 val(2,1) = (cth * sphi);
 val(2,2) = (cth * cphi);
 val(2,3) = z;
 val(3,0) = 0.0;
 val(3,1) = 0.0;
 val(3,2) = 0.0;
 val(3,3) = 1.0;

 return *this;

}

matrix & matrix::DHMatrix(double cosrotate, double sinrotate,
            double costwist, double sintwist,
            double length, double translate)
{
 val(0,0) = cosrotate;
 val(0,1) = -1 * sinrotate;
 val(0,2) = 0.0;
 val(0,3) = length;
 val(1,0) = sinrotate * costwist;
 val(1,1) = costwist * cosrotate;
 val(1,2) = -1 * sintwist;
 val(1,3) = translate * -1 * sintwist;
 val(2,0) = sintwist * sinrotate;
 val(2,1) = sintwist * cosrotate;
 val(2,2) = costwist;
 val(2,3) = translate * costwist;
 val(3,3) = 1.0;
```

```
 return *this;
}
Matrix & matrix::UpdateTMatrix(double a, double b,double c, double d)
{
 a = a * deg_to_rad;
 b = b * deg_to_rad;
 double cosrotate = cos(a);
 double sinrotate = sin(a);
 double costwist = cos(b);
 double sintwist = sin(b);
 DHMatrix(cosrotate, sinrotate, costwist, sintwist,c,d);

 return *this;
}

matrix & matrix::TransformNodeList(matrix &H_matrix, matrix &b)
{
 matrix temp(4,1,0.0);
 for (int i = 0; i<b.rows(); i++) {
              temp.val(0,0) = b.val(i,0);
              temp.val(1,0) = b.val(i,1);
              temp.val(2,0) = b.val(i,2);
              temp.val(3,0) = b.val(i,3);
              matrix  middle = H_matrix * temp;

  val(i,0) = middle.val(0,0);
  val(i,1) = middle.val(1,0);
  val(i,2) = middle.val(2,0);
  val(i,3) = middle.val(3,0);
 };

 return *this;
}

matrix & matrix::TransformBodyList(matrix &H_matrix, matrix &b)
{
 matrix temp(4,1,0.0);
 for (int i=0; i<b.rows(); i++) {
              temp.val(0,0) = b.val(i,0);
              temp.val(1,0) = b.val(i,1);
              temp.val(2,0) = b.val(i,2);
              temp.val(3,0) = b.val(i,3);
              matrix middle = H_matrix * temp;

              val(i,0) = middle.val(0,0);
              val(i,1) = middle.val(1,0);
              val(i,2) = middle.val(2,0);
              val(i,3) = middle.val(3,0);
 };
 return *this;
}

// ************************************************************
// FILENAME: MatrixMy.H
// PURPOSE: To provide for a matrix class to accomplish
//       some necessary robotic and kinematic needs.
// COMMENTS: DHMatrix, Homogeneous Transform, and
//    TransformNodeList are included
// ************************************************************

#ifndef H_MATRIX
#define H_MATRIX

const double deg_to_rad = .017453292519943295;

class matrix {
```

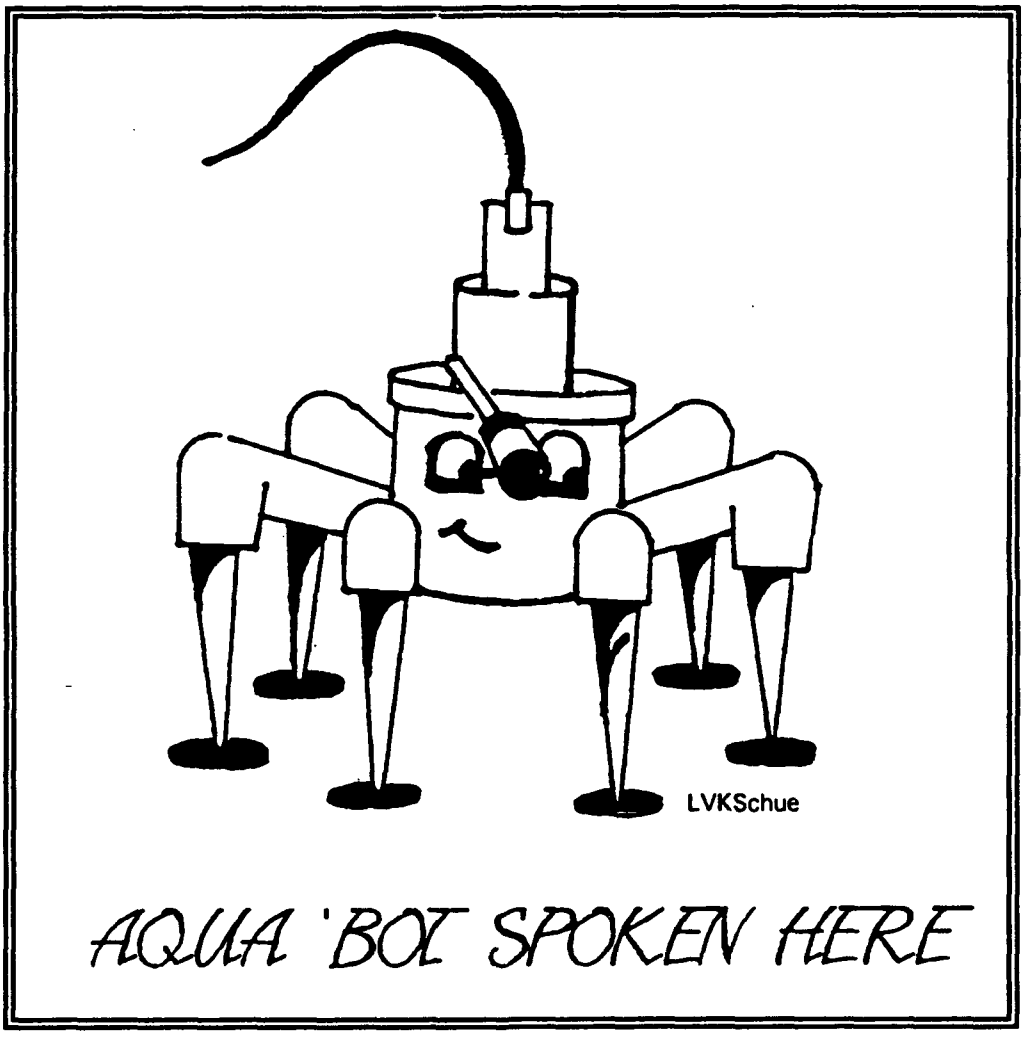
```
  struct matrep {
            double **m;
            int r, c, n;
  }*p;

public:
  matrix(const matrix& x);   //copy initializer
  ~matrix();
  matrix();
  matrix(int, int, double );
  matrix operator=(const matrix& rval);
  matrix operator+(const matrix& rval);
  matrix operator*(const matrix& rval);
  matrix operator*(double);
  double & val(int row, int col)const;
  void print();
  int rows() const {return p->r;};
  int cols() const {return p->c;};
  matrix & HomogeneousTransform(double,double,double,double,double,double);
  matrix & DHMatrix(double, double, double,double, double, double);
  matrix & UpdateTMatrix(double, double, double, double);
  matrix & TransformNodeList(matrix&, matrix&);
  matrix & TransformBodyList(matrix&, matrix&);
};

#endif
```

APPENDIX D: *AQUAROBOT* SIMULATOR USER'S MANUAL

This appendix contains the information necessary to use the version of the *AquaRobot* Simulator working at the time this thesis was written. The User's Guide was intended as a stand alone document; it may be removed from the thesis and used separately.



AQUAROBOT SIMULATOR USER'S GUIDE

AQUAROBOT SIMULATOR

USER'S GUIDE

Prepared by:

LT John Goetz, USN
Professor Yutaka Kanayama
Mr. Charles Lombardo, Staff
Professor Robert McGhee
Mr. Kenji Suzuki, Researcher
of the
Computer Science Department

and

LT Sandra Davidson, USN
LT Charles Schue, USCG
of the
Electrical and Computer Engineering Department

Naval Postgraduate School
Monterey, CA

June 1993

PREFACE

PURPOSE OF THE MANUAL

This manual provides user-specific information necessary to operate the *AquaRobot* Simulator on the Silicon Graphics Personal Iris. Code modification details are purposely excluded.

KNOWLEDGE LEVEL

Some knowledge of the Silicon Graphics workstation, the UNIX operating system, and the mouse pointing device is presumed.

CONTENTS

I. LOGGING ON

Step 1: At the *Login* screen, type in the appropriate responses to the *Login* prompts.

Login: aquarobo
Password: _____

The password is available from Dr. Kanayama or Dr. McGhee. Once *Login* is completed, the UNIX operating system is loaded.

Step 2: Click on the *CONSOLE* icon and size the *CONSOLE* window as desired.

Step 3: Press *enter* to get to the command prompt.

Step 4: Use the change directory command (*cd*) to change to the directory shown.

WorkSpace/thesis/arsim

Step 5: Check to ensure you are in the correct directory using the present work directory command (*pwd*). The correct directory is as shown.

/n/auvsim4/work/aquarobo/WorkSpace/thesis/arsim

Step 6: You have now completed the *Login* process and are positioned in the correct working directory.

II. COMPILING

Step 7: To compile this version program, type in the **make** command (*make*) and press *enter*. The **make** command has executed properly when the command prompt returns. Once the command prompt returns, the program has compiled correctly.

III. USING THE SIMULATOR

Step 8: At the command prompt, type **aqua** to start the AquaRobot Simulator. Some interaction between the user and the simulator is now required. When prompted, determine whether you want to see the discrete (type in 0) or continuous (type in 1) alternating tripod gaits. Then determine the desired goal point and enter its x-coordinate, space, and y-coordinate. The positive x-axis lies lengthwise to the right on the screen and the positive y-axis is outward towards the viewer. Next, you may be prompted to select the footpad size for the simulation. Choosing zero (0) selects the 25 centimeter pootpad; choosing one (1) selects the 45 centimeter footpad. The *CONSOLE* window shows program status and the *AQUAROBOT* window shows the graphic output.

Step 9: Four viewpoints are available to the user as a right mouse button menu: Camera: ABOVE, Camera: LEG5-6 VIEW, Camera: LEG1 VIEW, and Camera: LEG4 VIEW. To change the viewpoint from the default Camera: LEG2-3 VIEW view, quickly press the right mouse button after typing **aqua** and moving the mouse pointer into the *AQUAROBOT* window.

NOTE: The menu selections FILEREAD, RESETFILE, and KEYBDREAD should not be used.

Step 10: The RESET menu selection places AquaRobot in the default LEG2-3 VIEW.

Step 11: To repeat the **aqua** program, with the default Camera: LEG2-3 VIEW viewpoint, move the mouse pointer into the *CONSOLE* window and choose one of the following methods:

a. type **aqua**
b. press *enter*
type *!!* (this is the history/repeat sequence)
press *enter*

Step 12: To exit the **aqua** program, press the right mouse menu button and select Exit from the choices listed.

Step 13: To exit the *CONSOLE* window, use the right mouse button to get Log Out; then answer "yes".

ATTACHMENT A: FLOWCHART and FUNCTION DESCRIPTIONS

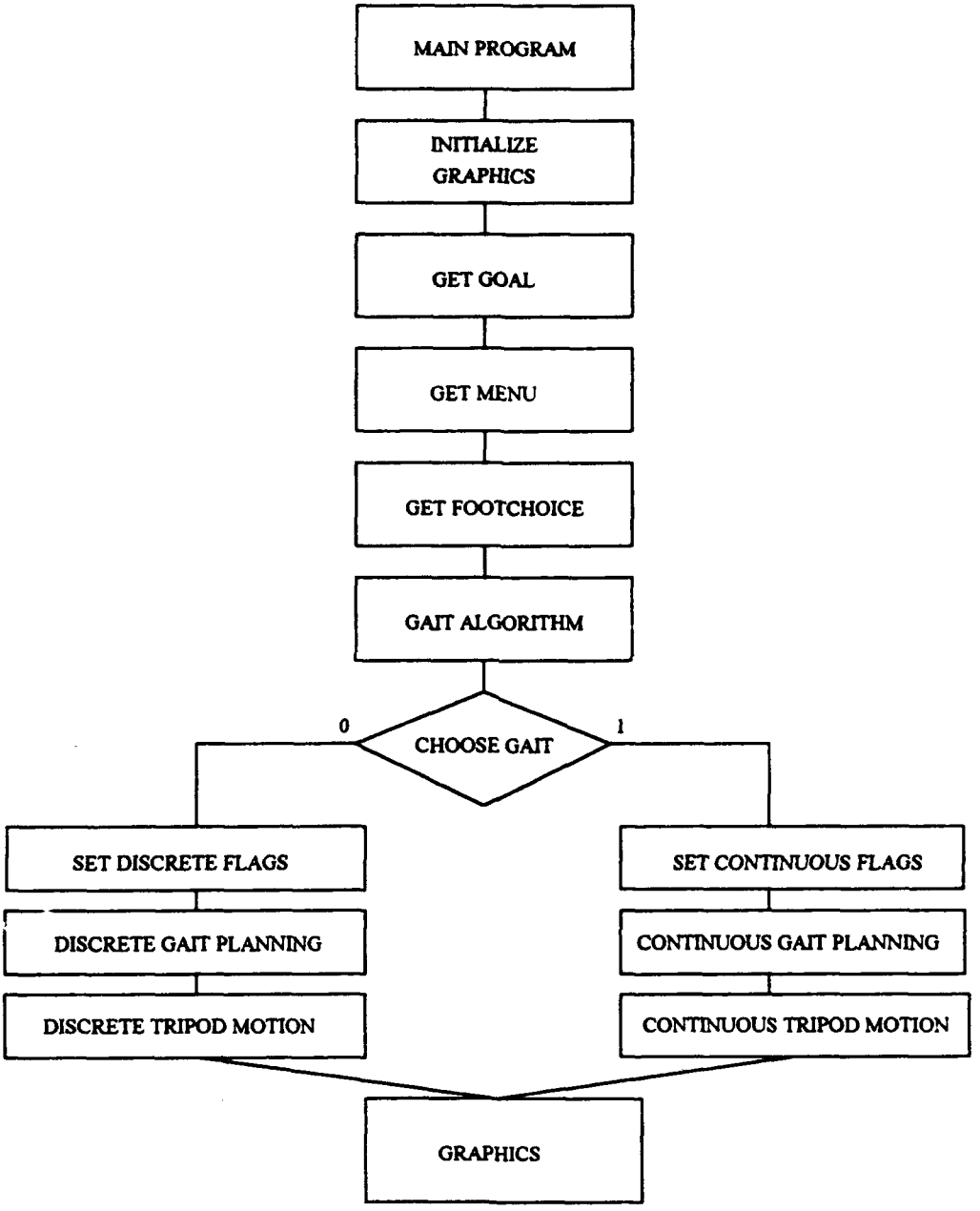


Figure D-1. AquaRobot Simulator Flow Chart.

The simulator files are separated into four modules. The files that comprise the Makefile, Graphics, and Matrix Manipulation Modules are only cursorily addressed here. The Gait Planning Module is described in more detail.

1. Makefile

The *Makefile* allows the *make* utility to intelligently compile the program. This file includes instructions for how and when to compile the various files that comprise the *AquaRobot* Simulator. The *AquaRobot* simulation program uses the UNIX *make* utility to link together the 27 individual files with the graphics, math, and standard input-output libraries. In this case, the Makefile generates the executable program *aqua*.

2. Graphics

The 3D stick-figure graphics code was originally written by Sandra Davidson [DAV93]. The graphics files (code modules) are included in Appendix C for easier reference. Some file names were changed to avoid software configuration management problems. Additionally, some graphics code was modified because of the requirements of the gait planning code. For example, the FARCLIPPING value and the CAMERA viewing angles were changed in the *bot.H* file and the CB height was changed in the *AbBody.C* file. Because the Graphics Module polls the Gait Planning Module, a suitable interface between the two was constructed in the *Kinematics.H* and *Kinematics.C* files. The graphics code consists of the following files:

- *AbBody.H* and *AbBody.C*;
- *AbLeg.H* and *AbLeg.C*;
- *AbRigid.H* and *AbRigid.C*;
- *bot.H* and *bot.C*;
- *Kinematics.H* and *Kinematics.C*;

- *Link.H* and *Link.C*;
- *Link0.H* and *Link0.C*;
- *Link1.H* and *Link1.C*;
- *Link2.H* and *Link2.C*; and
- *Link3.H* and *Link3.C*.

3. Matrix Manipulation

The matrix manipulation code was also originally written by Sandra Davidson [DAV93]. It provides 4 x 4 matrix multiplication capability. This code is an essential part of the graphics code, although it is not used in the gait planning code. The Matrix Manipulation Module includes the following files, found in Appendix C:

- *MatrixMy.H* and *MatrixMy.C*.

4. Gait Planning

The Gait Planning Module (GPM) consists of the following files:

- *arconst.H;*
- *arfunc.H* and *arfunc.C*; and
- *artpgait.H* and *artpgait.C*.

To support easier reference and maintenance, most constants are grouped together into the file *arconst.H* (aquarobot **constants**).

The *arfunc.H/C* (aquarobot **functions**) files contain 13 functions. The *min* and *max* functions simply return the minimum or maximum of two expressions, respectively. The *ellipse* function calculates the incremental foot trajectory along an elliptical path between the current foot position and the desired foot position. The *kinematics* function performs kinematics for the Gait Planning Module. Kinematics for the Graphics Module are performed in the Graphics *Kinematics.C/H* files. The *inv_kinematics* function performs the inverse kinematics operations. The *world_body* function provides coordinate

transformation from the body-fixed coordinate system to the world coordinate system. The *body_world* function provides coordinate transformation from the world coordinate system to the body-fixed coordinate system.

The *maxdistance_25cm* function determines the maximum stride for the workspace defined by the 25 centimeter footpad. The *maxdistance_45cm* function determines the maximum stride for the workspace defined by the 45 centimeter footpad. The *arcint* function calculates the intersection of a directed line and an arc segment. The *segint* function calculates the intersection of a directed line and a line segment. The *stable* function calculates the stability of a tripod. The staparam function determines the longitudinal stability margin and the x and y intercepts of the directional ray originating at the CB and terminating at the point of intersection on the selected tripod.

The *artpgait.H/C* (**aquarobot tripod gait**) files 15 functions. The *get_goal* function allows the user to input the desired goal point to which the *AquaRobot* simulation walks. The *get_menu* function allows the user to choose between the DATG or the CATG. The *get_footchoice* function allows the user to determine whether the 25 centimeter or 45 centimeter footpad is used. The *gait_algorithm* function executes either the DATG or the CATG gait algorithms, as previously determined by the user. The *robot_model* function initializes the joint angles for the simulation.

The *disc_set_flag* function initializes the flags used during the DATG algorithm. The *disc_gait_planning* function determines the walking parameters, such as stride length and direction, used during the DATG algorithm. The *disc_tripod_motion* function calculates and executes incremental joint motions necessary to move the *AquaRobot* simulation using the DATG.

The *cont_set_flag* function initializes the flags used during the CATG algorithm. The *cont_gait_planning* function determines the walking parameters, such as stride length

and direction, used during the CATG algorithm. The *cont_tripod_motion* function calculates and executes the incremental joint motions necessary to move the *AquaRobot* simulation using the CATG.

Some functions in the *artpgait.C* file are used only for printing out the status of joints, feet, etc. These functions include *print_status*, *print_walkpara*, *print_gnu_data*, and *print_joint_data*.

ATTACHMENT B: REQUIRED FILES LISTING

Makefile

Makefile

Graphics

AbBody.H
AbBody.C
AbLeg.H
AbLeg.C
AbRigid.H
AbRigid.C
bot.H
bot.C
Kinematics.H
Kinematics.C
Link.H
Link.C
Link0.H
Link0.C
Link1.H
Link1.C
Link2.H
Link2.C
Link3.H
Link3.C

Matrix Manipulation

MatrixMy.C
MatrixMy.H

Gait Planning

arconst.H
arfunc.H
arfunc.C
artpgait.H
artpgait.C

LIST OF REFERENCES

[MCG85] McGhee, R.B., "Vehicular Legged Locomotion," *Advances in Automation and Robotics*, Vol. 1, Jai Press, Inc., pp. 259-284, 1985.

[ISH83] Ishino, Y., Naruse, T., Sawano, T., and Honma, N., "Walking Robot for Underwater Construction," *International Conference on Advanced Robotics*, pp. 107-114, 1983.

[AKI89] Akizono, J., Iwasaki, M., Nemoto, T., and Asakura, O., "Development on Walking Robot for Underwater Inspection," *Advanced Robotics 1989: Proceedings of the 4th International Conference on Advanced Robotics*, Springer-Verlag, pp. 652-663, June 1989.

[IWA87] Report of the Port and Harbour Research Institute, Japan, Vol. 26, No. 5, *Development on Aquatic Walking Robot for Underwater Inspection*, by M. Iwasaki and others, pp. 393-422, December 1987.

[IWA88a] Iwasaki, M., et al., "Development on Aquatic Walking Robot for Underwater Construction," *Proceedings of the 5th International Symposium on Robotics in Construction*, JSCE, Vol. 2, Tokyo, Japan, pp. 765-774, June 6-8, 1988.

[IWA88b] Iwasaki, M., et al., "Development on Aquatic Walking Robot for Underwater Inspection," *Proceedings of the USA-Japan Symposium on Flexible Automation*, pp. 659-664, 1988.

[IWA90] Iwasaki, M., Akizono, J., Nemoto, T., and Asakura, O., "Field Test of Aquatic Walking Robot for Undersea Inspection," *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, CA, pp. 109-112, October 23-26, 1990.

[CRA86] Craig, J.J., *Introduction to Robotics*, Second Edition, Addison-Wesley Publishing Company, Inc., 1986.

[HIR84] Hirose, S., "A Study of Design and Control of a Quadruped Walking Vehicle," *International Journal of Robotics Research*, Vol. 3, No. 2, pp. 113-133, Summer 1984.

[HIR86a] Hirose, S., Fubuda, Y., and Kikuchi, H., "The Gait Control System of a Quadruped Walking Vehicle," *Advanced Robotics*, Vol. 1, No. 4, pp. 289-323, December 1986.

[HIR86b] Hirose, S., Kikuchi, H., and Umetani, Y., "The Standard Circular Gait of a Quadruped Walking Vehicle," *Advanced Robotics*, Vol. 1, No. 2, pp. 143-164, 1986.

[HIR88] Hirose, S., and Yokoi, K., "The Standing Posture Transformation Gait of a Quadruped Walking Vehicle," *Advanced Robotics*, Vol. 2, No. 4, pp. 345-359, 1988.

[MCG79] McGhee, R.B., and Iswandhi, G.I., "Adaptive Locomotion of a Multilegged Robot Over Rough Terrain," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-9, No. 4, pp. 176-182, April 1979.

[SON89] Song, S., and Waldron, K.J., *Machines That Walk: The Adaptive Suspension Vehicle*, The MIT Press, 1989.

[TOD85] Todd, D.J., *Walking Machines*, Kogan Page Ltd., 1985.

[LYM87] Lyman, R.L., *A Computer Simulation Study of Tripod Follow-The-Leader Gait Coordination for a Hexapod Walking Machine*, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1987.

[KWA90] Kwak, S.H., and McGhee, R.B., "Rule-based Motion Coordination for a Hexapod Walking Machine," *Advanced Robotics*, Vol. 4, No. 3, pp. 263-282, December 1990.

[LEE88a] Lee, W., and Orin, D.E., "The Kinematics of Motion Planning for Multilegged Vehicles Over Uneven Terrain," *IEEE Journal of Robotics and Automation*, Vol. 4, No. 2, pp. 204-212, April 1988.

[LEE88b] Lee, W., and Orin, D.E., "Omnidirectional Supervisory Control of a Multilegged Vehicle Using Periodic Gaits," *IEEE Journal of Robotics and Automation*, Vol. 4, No. 6, pp. 635-642, December 1988.

[WAL84] Waldron, K.J., Vohnout, V.J., Pery, A., and McGhee, R.M., "Configuration Design of the Adaptive Suspension Vehicle," *International Journal of Robotics Research*, Vol. 3, No. 2, pp. 37-48, 1984.

[DAV93] Davidson, S.L., *An Experimental Comparison of CLOS and C++ Implementations of an Object-Oriented Graphical Simulation of Walking Robot Kinematics*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1993.

[GRI93] Grim, C.J., *An Object Oriented Program Specification for a Mobile Robot Motion Control Language*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1993.

[KAN92] Kanayama, Y., Lecture notes for CS4313 (Advanced Robotics), Naval Postgraduate School, 1992 (unpublished).

[ODE86] Anon., *Annual Report*, Odetics, Inc., Corporate Headquarters, 1515 S. Manchester Ave., Anaheim, CA, 92802-2907, March 1986.

[MCG86] McGhee, R.B., "Computer Coordination of Motion for Omni-directional Hexapod Walking Machines," *Advanced Robotics*, Vol. 1, No. 2, pp. 91-99, 1986.

[MCG72] McGhee, R.B., and Jain, A.K., "Some Properties of Regularly Realizable Gait Matrices," *Mathematical Biosciences*, Vol. 13, No. 1, pp. 179-193, February 1972.

[MCG68a] McGhee, R.B., "Some Finite State Aspects of Legged Locomotion," *Mathematical Biosciences*, Vol. 2, No. 1, pp. 67-84, February 1968.

[TAK93] Minutes of AquaRobot monthly meeting, H. Takahashi (PHRI); Y. Kanayama (NPS), R. McGhee (NPS), C. Schue (NPS), S. Davidson (NPS), C. Lombardo (NPS), and K. Suzuki (University of Electrocommunications) attending, January 21, 1993.

[DEN55] Denavit, J., and Hartenberg, R.S., "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," *Journal of Applied Mechanics*, pp. 215-221, June 1955.

[DON87] Donner, M.D., *Real-Time Control of Walking*, Birkhäuser Boston, 1987.

[PHR93] Interview between M. Iwasaki, H. Takahashi, and S. Shiraiwa, Port and Harbor Research Institute, Yokosuka, Japan, and the author, April 12-16, 1993.

BIBLIOGRAPHY

Cooper, C.R., *Introduction to Matlab*, EC1010 Course Notes, Naval Postgraduate School, Monterey, CA, July 1992.

Davidson, S.L., *An Experimental Comparison of CLOS and C++ Implementations of an Object-Oriented Graphical Simulation of Walking Robot Kinematics*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1993.

Finney, R.L., and Thomas, G.B., *Calculus*, Addison-Wesley Publishing Company, Inc., 1990.

Grim, C.J., *An Object Oriented Program Specification for a Mobile Robot Motion Control Language*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1993.

IRIS User's Guide, Silicon Graphics, Inc., Mountain View, CA, 1986.

Leendert, A., *C++ for Programmers*, John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex, England, 1991.

Lyman, R.L., *A Computer Simulation Study of Tripod Follow-The-Leader Gait Coordination for a Hexapod Walking Machine*, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1987.

Naval Postgraduate School Proposal to National Science Foundation, *Computer Simulation and Control of Autonomous Underwater Walking Robots*, by Y. Kanayama and R.B. McGhee, December 1990.

PC-Matlab Tutorial, The MathWorks, Inc., Sherborn, MA, June 1987.

Purdum, J., *PC Magazine Guide to C Programming*, Ziff-Davis Press, Emeryville, CA, 1992.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor Yutaka Kanayama, Code CS/Ka Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Professor Roberto Cristi, Code EC/Cx Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Professor Robert B. McGhee, Code CS/Mz Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	1
7.	Mr. Norman Caplan Biological and Critical Systems Divisions Engineering Directorate National Science Foundation 1800 G Street, NW Washington, DC 20550	1

8. Mr. Hidetoshi Takahashi — 1
Port and Harbor Research Institute
Ministry of Transport
1-1, 3-Chome, Nagase
Yokosuka, Japan

9. Commandant (G-TP-2) — 1
U.S. Coast Guard
2100 2nd Street, SW
Washington, DC 20593-0001

10. Commandant (G-PIM-2/O) — 1
U.S. Coast Guard
2100 2nd Street, SW
Washington, DC 20593-0001

11. Commanding Officer (nlr) — 2
U.S. Coast Guard EECEN
Attn: LT Charles Schue
Wildwood, NJ 08260-0060